Agilent Technologies

SystemVue 2010.07
2010
Algorithm Design Library

Adaptive Equalizers

Adaptive Equalizer Parts provide a set of adaptive filtering algorithms and other parts which can be used to develop channel equalizers. Equalizers are used in communication systems to compensate for the effect of multi-path channels. Although Adaptive Equalizer Parts is designed with equalization as its main application, the algorithms are provided in a form which makes them suitable for other adaptive filtering applications. The following adaptive algorithms are provided:

- Least Mean Squares (LMS)
- Affine Projection Algorithm (APA)
- Recursive Least Squares (RLS)
- QR Recursive Least Squares (QR-RLS)

These algorithms can be combined with other parts from Algorithm Design/Adaptive Equalizer Parts and SystemVue to design, simulate and analyze advanced channel equalizer subsystems. For more information see *About Adaptive Equalizer Parts* (algorithm)

Contents

About Adaptive Equalizer Parts

Introduction

Algorithm Design/Adaptive Equalizer Parts provide a set of adaptive filtering algorithms and other parts which can be used to develop channel equalizers. Equalizers are used in communication systems to compensate for the effect of multi-path channels. Although Algorithm Design/Adaptive Equalizer Parts is designed with equalization as its main application, the algorithms are provided in a form which makes them suitable for other adaptive filtering applications. The following adaptive algorithms are provided:

- Least Mean Squares (LMS)
- Affine Projection Algorithm (APA)
- Recursive Least Squares (RLS)
- QR Recursive Least Squares (QR-RLS)

These algorithms can be combined with other parts from Algorithm Design/Adaptive Equalizer Parts and SystemVue to design, simulate and analyze advanced channel equalizer subsystems.

Adaptive Filtering

The diagram below shows an adaptive filter and its associated input and output signals. The aim of any adaptive filter is to minimize the error signal *e*[*k*], which represents the difference between the desired signal *d*[*k*] and the output of the adaptive filter *y*[*k*].

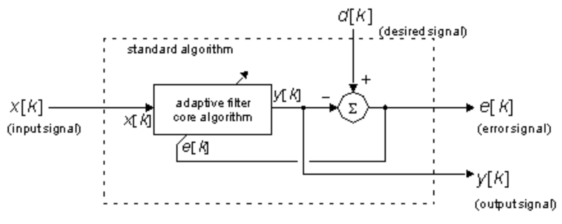


Figure 1: Adaptive Filtering

The adaptive algorithms provided with Algorithm Design/Adaptive Equalizer Parts come in two forms; **standard** parts and **core** parts. This is illustrated in the above diagram.

Standard Parts

The standard parts take *x*[*k*] and *d*[*k*] as inputs and produce *e*[*k*] and *y*[*k*] as outputs. Therefore the standard parts implement all of the functionality shown in the adaptive filter diagram above.

Core Parts

The core parts take *x*[*k*] and *e*[*k*] as inputs and produce *y*[*k*] as output. Therefore the core parts require the user to calculate *e*[*k*] externally and feed this back into the part. The core parts also perform their internal operations in a different order to take this feedback into account. The core parts make it possible to build more complicated equalizers such as Decision Feedback Equalizers (DFEs) and blind equalizers.

Equalization

In adaptive filtering, the relationship between *y*[*k*] and *d*[*k*] determines the mode or configuration of the adaptive filter. The diagram below shows an adaptive filter in "inverse system identification" mode where the adaptive filter can be said to "equalize" an unknown system. In this example the adaptive filter is being used to equalize a communication channel by setting *d*[*k*] = *I*[*k*-*u*] where *u* is a delay introduced to ensure that the equalization filter is causal. In practice a training sequence known by both the transmitter and receiver can be used to create both *I*[*k*] and *d*[*k*]. Alternatively *d*[*k*] can be derived from a non-linear function of *y*[*k*] at the receiver as in the case of blind and decision directed equalizers.

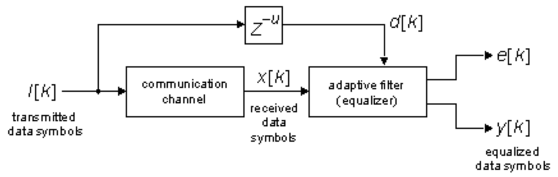


Figure 2: Equalization

Algorithms

Definitions

The adaptive algorithms in Algorithm Design/Adaptive Equalizer Parts are defined in matrix notation, therefore this section introduces the definitions which are common to all algorithms. The diagram below shows an adaptive filter with *N* feed forward weights and *M* feed back weights. For *M=0* the filter is FIR (Finite Impulse Response). For *M>0* the filter is IIR (Infinite Impulse Response).

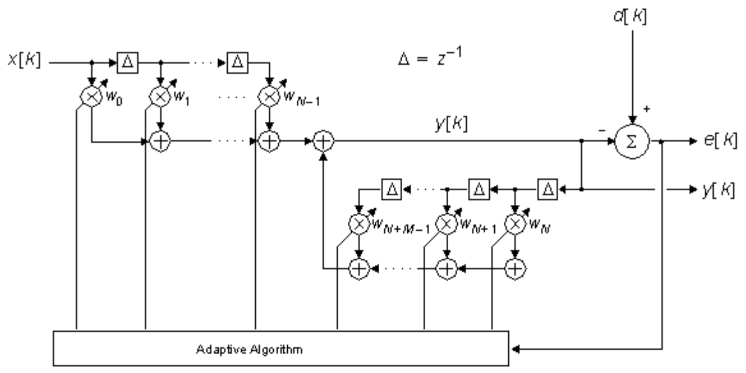


Figure 3: FIR / IIR Adaptive Filter

The input and output data sample vectors corresponding to the input and output delay lines are

$$\mathbf{x}[k] = [x[k]\ x[k-1]\ \dots\ x[k-N+1]]^T$$
$$\mathbf{y}[k-1] = [y[k-1]\ y[k-2]\ \dots\ y[k-M]]^T$$

where [ ]^T represents the transpose of a vector. It is convenient to concatenate these vectors as follows

$$\mathbf{z}[k] = [x[k]\ x[k-1]\ \dots\ x[k-N+1]\ y[k-1]\ y[k-2]\ \dots\ y[k-M]]^T$$

The feed forward and feedback weight vectors are defined as

$$\mathbf{w}_{ff} = [w_0\ w_1\ \dots\ w_{N-1}]^T$$
$$\mathbf{w}_{fb} = [w_N\ w_{N+1}\ \dots\ w_{N+M-1}]^T$$

It is also convenient to combine these into a single vector

$$\mathbf{w} = [\mathbf{w}_{ff}\ |\ \mathbf{w}_{fb}]^T = [w_0\ w_1\ \dots\ w_{N-1}\ w_N\ \dots\ w_{N+M-1}]^T$$

The output of the adaptive filter is

$$\begin{aligned} y[k] &= \mathbf{w}_{ff}^H[k-1]\mathbf{x}[k] + \mathbf{w}_{fb}^H[k-1]\mathbf{y}[k-1] \\ &= \mathbf{w}^H[k-1]\mathbf{z}[k] \end{aligned}$$

where [ ]^H represents the Hermitian of a vector (transpose and complex conjugate). Finally, the error signal *e*[*k*] is defined as

$$\begin{aligned} e[k] &= d[k] - y[k] \\ &= d[k] - \mathbf{w}^H[k-1]\mathbf{z}[k] \end{aligned}$$

LMS

The Least Mean Squares (LMS) algorithm is a *stochastic gradient technique* which tries to minimize the Mean Square Error (MSE); that is it tries to minimize E{$e[k]$} where E{} is the expectation operator. On each weight update, the LMS algorithm estimates the direction of steepest descent based on the current state vector $\boldsymbol{z}[k]$ and the current value of the desired signal $d[k]$. To allow for different step sizes in the feed forward and feedback sections of the filter we shall define the LMS explicitly in terms of these sections; that is we refer to $\boldsymbol{w}_{ff}[k]$, $\boldsymbol{w}_{fb}[k]$, $\boldsymbol{x}[k]$ and $\boldsymbol{y}[k\text{-}1]$ rather than $\mathbf{w}[k]$ and $\boldsymbol{z}[k]$. The LMS algorithm comprises the following steps.

1. Compute filter output
   $y[k] = w_{ff}^{H}[k]x[k] + w_{fb}^{H}[k]y[k-1]$
2. Compute error signal
   $e[k] = d[k] - y[k]$
3. Update feed forward and feedback weights
   $w_{ff}[k+1] = w_{ff}[k] + 2\mu_{ff}x[k]e^{*}[k]$

   $w_{fb}[k+1] = w_{fb}[k] + 2\mu_{fb}y[k-1]e^{*}[k]$

where μ_{ff} and μ_{fb} are the feed forward and feedback step sizes respectively. The Normalized Least Mean Squares (NLMS) algorithm is a variation of the LMS which normalizes the step sizes according to the energy in $\boldsymbol{z}[k]$. The NLMS weight update is defined as follows

$$w_{ff}[k+1] = w_{ff}[k] + 2\frac{\mu_{ff}}{z^{H}[k]z[k]}x[k]e^{*}[k]$$

$$w_{fb}[k+1] = w_{fb}[k-1] + 2\frac{\mu_{fb}}{z^{H}[k]z[k]}y[k-1]e^{*}[k]$$

where $()^{*}$ indicates the complex conjugate.

The Non-Canonical LMS (NC-LMS) is a modified version of the LMS algorithm which works with the transpose form of an FIR filter. The transpose FIR has a broadcast line instead of a tapped delay line and delays between the adders in the summation chain as shown in the diagram below. Despite this, it has exactly the same functionality as a canonical FIR filter. The non-canonical FIR structure has advantages for hardware implementation as the delays form a natural pipeline through the summation chain. However the LMS algorithm cannot be applied as it would require time advances to distribute the error signal to the weights for the purpose of updating (i.e. it would be non-causal).

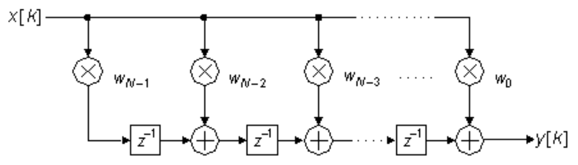


Figure 4: Transpose FIR Filter

The Non-Canonical LMS (NC-LMS) is shown in the diagram below. It is important to note that the NC-LMS has different behavior to that of the standard LMS. In particular it tends to be less stable and slower to converge.

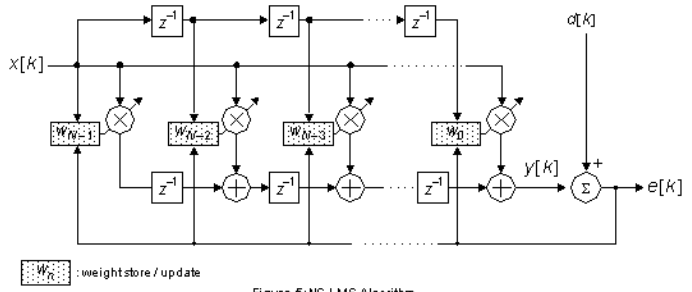


Figure 5: NC-LMS Algorithm

It can be shown that the NC-LMS actually implements the following weight update equation.

$$\mathbf{w}_{ff}[k+1] = \mathbf{w}_{ff}[k] + 2\mu_{ff}\begin{bmatrix} e[k]x[k] \\ e[k-1]x[k-2] \\ \dots \\ e[k-N+1]x[k-2(N-1)] \end{bmatrix}$$

Currently the LMS algorithm in Algorithm Design/Adaptive Equalizer Parts only supports a transpose filter structure in the feed forward section. The feedback part is implemented in standard form therefore always uses the standard LMS algorithm.

APA

The Affine Projection Algorithm (APA) is also a stochastic gradient technique which tries to minimize the mean square error E{*e*[*k*]}. The APA algorithm estimates the direction of steepest descent based on the *p* most recent state vectors {***z***[*k*], ***z***[*k*-1] ... ***z***[*k*-*p*+1]} and the *p* most recent samples of the desired signal {*d*[*k*], d[*k*-1]... d[*k*-*p*+1]}. The parameter *p* is known as the *order* of the algorithm. The following vector and matrix definitions are required before stating the computation steps of the APA algorithm.

$$\begin{aligned} \mathbf{d}[k] &= \left[d[k]\ d[k-1]\ \dots\ d[k-p+1]\right]^{T} \\ Z[k] &= \left[z[k]\ z[k-1]\ \dots\ z[k-p+1]\right] \\ X[k] &= \left[x[k]\ x[k-1]\ \dots\ x[k-p+1]\right] \\ Y[k-1] &= \left[\mathbf{y}[k-1]\ \mathbf{y}[k-2]\ \dots\ \mathbf{y}[k-p]\right] \\ \mathbf{R}[k] &= Z^{H}[k]Z[k] + \delta \mathbf{I} \end{aligned}$$

where ***I*** is the *p*-by-*p* identity matrix and δ is a *bias* factor introduced to prevent ***R***[*k*] from becoming singular. The APA algorithm then comprises the following steps:

1. Compute the filter output
   $y[k] = \mathbf{w}_{ff}^{H}[k]\mathbf{x}[k] + \mathbf{w}_{fb}^{H}[k]\mathbf{y}[k-1]$
2. Compute the error signal vector based on the current weights (note that ***e***[*k*] does not contain past samples of the error signal; it contains what the past error samples would have been given the current filter weights)
   $\mathbf{e}[k] = \mathbf{d}[k] - Z^{T}[k]\mathbf{w}^{*}[k]$
3. Update the filter weights
   $\mathbf{w}_{ff}[k+1] = \mathbf{w}_{ff}[k] + 2\mu_{ff}X[k]\mathbf{R}^{-1}[k]\mathbf{e}^{*}[k]$

   $\mathbf{w}_{fb}[k+1] = \mathbf{w}_{fb}[k] + 2\mu_{fb}Y[k-1]\mathbf{R}^{-1}[k]\mathbf{e}^{*}[k]$

where μ_{ff} and μ_{fb} are the feed forward and feedback step sizes respectively. Note that for *p=0* the APA is equivalent to the LMS algorithm.

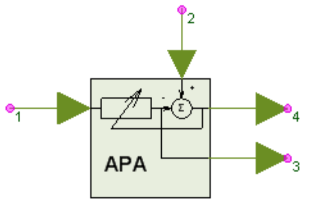
AdptFltAPA_Cx Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AdptFltAPA_Cx (algorithm)	Complex APA Adaptive Filter

AdptFltAPA_Cx



Description: Complex APA Adaptive Filter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltAPA Cx Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedfowardFilterStepSize	Step size of the algorithm for the EqLib coefficients.	0.4		Float	NO	(0:1)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
FeedbackFilterStepSize	Step size of the algorithm for the feedback stage.	0.4		Float	NO	(0:1)
Order	Selects the order of the APA algorithm.	3		Integer	NO	(1:∞)
Bias	Bias factor of the APA algorithm.	1E-08		Float	NO	(-∞:∞)
InitCoef	Initialized feed forward filter coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoefValue	Sets the feed forward filter coefficients to an inital value (used when InitCoef = YES).	[0.0+j*0.0]		Complex array	NO	[-∞:∞]

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	complex	NO
2	desiredSignal	complex	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	complex	NO
4	errorSignal	complex	NO

Notes/Equations

1. This model implements a complex arithmetic adaptive filter (FIR or IIR) using the Affine Projection Algorithm (APA). The APA algorithm tries to minimize the Mean Square Error (MSE) between the desired signal and the output of the filter.
2. During each iteration, the APA algorithm approximates the best direction in which to adjust the filter weights based on the most recent *p* state vectors and the most recent *p* samples of the desired signal where *p* is the algorithm order. This approximation is then used to update the filter weights. Part of this algorithm requires a *p*-by-*p* matrix inversion to be computed. The *Bias* parameter is added to the diagonal elements of the matrix prior to inversion in case it is singular.
3. For a detailed description of the APA algorithm see *About Adaptive Equalizer Parts* (algorithm).
4. See also *AdptFltAPA* (algorithm), *AdptFltCoreAPA* (algorithm), *AdptFltCoreAPA_Cx* (algorithm).

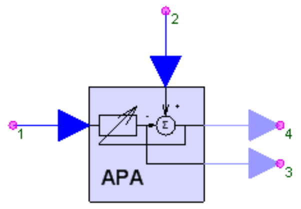
AdptFltAPA Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AdptFltAPA (algorithm)	APA Adaptive Filter

AdptFltAPA



Description: APA Adaptive Filter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltAPA Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedfowardFilterStepSize	Step size of the algorithm for the EqLib coefficients.	0.4		Float	NO	(0:1)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
FeedbackFilterStepSize	Step size of the algorithm for the feedback stage.	0.4		Float	NO	(0:1)
Order	Selects the order of the APA algorithm.	3		Integer	NO	(1:∞)
Bias	Bias factor of the APA algorithm.	1E-08		Float	NO	(-∞:∞)
InitCoef	Initialized feed forward filter coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoefValue	Sets the feed forward filter coefficients to an inital value (used when InitCoef = YES).	[0.0]		Floating point array	NO	[-∞:∞]

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	real	NO
2	desiredSignal	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	real	NO
4	errorSignal	real	NO

Notes/Equations

1. This model implements a real arithmetic adaptive filter (FIR or IIR) using the Affine Projection Algorithm (APA). The APA algorithm tries to minimize the Mean Square Error (MSE) between the desired signal and the output of the filter.
2. During each iteration, the APA algorithm approximates the best direction in which to adjust the filter weights based on the most recent *p* state vectors and the most recent *p* samples of the desired signal where *p* is the algorithm order. This approximation is then used to update the filter weights. Part of this algorithm requires a *p*-by-*p* matrix inversion to be computed. The *Bias* parameter is added to the diagonal elements of the matrix prior to inversion in case it is singular.
3. For a detailed description of the APA algorithm see *About Adaptive Equalizer Parts* (algorithm).
4. See also *AdptFltAPA_Cx* (algorithm), *AdptFltCoreAPA* (algorithm), *AdptFltCoreAPA_Cx* (algorithm).

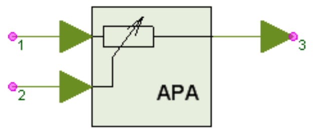
AdptFltCoreAPA_Cx Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AdptFltCoreAPA_Cx (algorithm)	Complex APA Adaptive Filter Core

AdptFltCoreAPA_Cx



Description: Complex APA Adaptive Filter Core
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltCoreAPA Cx Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedfowardFilterStepSize	Step size of the algorithm for the EqLib coefficients.	0.4		Float	NO	(0:1)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
FeedbackFilterStepSize	Step size of the algorithm for the feedback stage.	0.4		Float	NO	(0:1)
Order	Selects the order of the APA algorithm.	3		Integer	NO	(1:∞)
Bias	Bias factor of the APA algorithm.	1E-08		Float	NO	(-∞:∞)
InitCoef	Initialized feed forward filter coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoefValue	Sets the feed forward filter coefficients to an inital value (used when InitCoef = YES).	[0.0+j*0.0]		Complex array	NO	[-∞:∞]

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	complex	NO
2	errorSignal	complex	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	complex	NO

Notes/Equations

1. This model implements the core of a complex arithmetic adaptive filter (FIR or IIR) using the Affine Projection Algorithm (APA). The APA core tries to minimize the mean square of the error signal by adjusting its internal filter coefficients.
2. During each iteration, the APA algorithm approximates the best direction in which to adjust the filter weights based on the most recent *p* state vectors and the most recent *p* samples of the error signal, where *p* is the algorithm order. This approximation is then used to update the filter weights. Part of this algorithm requires a *p*-by-*p* matrix inversion to be computed. The *Bias* parameter is added to the diagonal elements of the matrix prior to inversion in case it is singular.
3. For a detailed description of the APA algorithm see *About Adaptive Equalizer Parts* (algorithm).
4. See also *AdptFltAPA* (algorithm), *AdptFltAPA_Cx* (algorithm), *AdptFltCoreAPA* (algorithm).

Note that since this is a core part, the user must generate the error signal externally and feed it back into the part. This means that the weight update is done on the time step after the output is calculated.

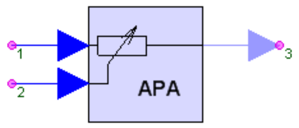
AdptFltCoreAPA Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AdptFltCoreAPA (algorithm)	APA Adaptive Filter Core

AdptFltCoreAPA



Description: APA Adaptive Filter Core
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltCoreAPA Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedfowardFilterStepSize	Step size of the algorithm for the EqLib coefficients.	0.4		Float	NO	(0:1)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
FeedbackFilterStepSize	Step size of the algorithm for the feedback stage.	0.4		Float	NO	(0:1)
Order	Selects the order of the APA algorithm.	3		Integer	NO	(1:∞)
Bias	Bias factor of the APA algorithm.	1E-08		Float	NO	(-∞:∞)
InitCoef	Initialized feed forward filter coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoefValue	Sets the feed forward filter coefficients to an inital value (used when InitCoef = YES).	[0.0]		Floating point array	NO	[-∞:∞]

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	real	NO
2	errorSignal	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	real	NO

Notes/Equations

1. This model implements the core of a real arithmetic adaptive filter (FIR or IIR) using the Affine Projection Algorithm (APA). The APA core tries to minimize the mean square of the error signal by adjusting its internal filter coefficients.
2. During each iteration, the APA algorithm approximates the best direction in which to adjust the filter weights based on the most recent *p* state vectors and the most recent *p* samples of the error signal, where *p* is the algorithm order. This approximation is then used to update the filter weights. Part of this algorithm requires a *p*-by-*p* matrix inversion to be computed. The *Bias* parameter is added to the diagonal elements of the matrix prior to inversion in case it is singular.
3. For a detailed description of the APA algorithm see *About Adaptive Equalizer Parts* (algorithm).
4. See also *AdptFltAPA* (algorithm), *AdptFltAPA_Cx* (algorithm), *AdptFltCoreAPA_Cx* (algorithm).

Note that since this is a core part, the user must generate the error signal externally and feed it back into the part. This means that the weight update is done on the time step after the output is calculated.

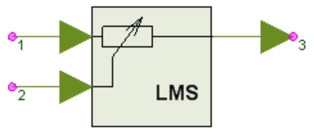
AdptFltCoreLMS_Cx Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AdptFltCoreLMS_Cx (algorithm)	Complex LMS Adaptive Filter Core

AdptFltCoreLMS_Cx



Description: Complex LMS Adaptive Filter Core
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltCoreLMS Cx Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedfowardFilterStepSize	Step size of the algorithm for the EqLib coefficients.	0.4		Float	NO	(0:1)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
FeedbackFilterStepSize	Step size of the algorithm for the feedback stage.	0.4		Float	NO	(0:1)
NLMS	Normalisation of the step size by the input signal power. If on the algorithm used is called NLMS.: OFF , ON	ON		Enumeration	NO	(0:1)
TransposeStructure	Transpose filter structure in the filter part of the adaptive structure.: OFF , ON	OFF		Enumeration	NO	(0:1)
Symmetric	Symmetric feedforward coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoef	Initialized feed forward filter coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoefValue	Sets the feed forward filter coefficients to an inital value (used when InitCoef = YES).	[0.0+j*0.0]		Complex array	NO	[-∞:∞]

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	complex	NO
2	errorSignal	complex	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	complex	NO

Notes/Equations

1. This model implements the core of a complex arithmetic adaptive filter (FIR or IIR) using the Least Mean Squares (LMS) or Non-Canonical Least Mean Squares (NC-LMS) algorithm. These algorithms try to minimize the mean square of the error signal by adjusting their internal filter coefficients.
2. During each iteration the LMS algorithm approximates the best direction in which to adjust the filter weights based on the current state vector and the current sample of the error signal. This estimate is then used to update the filter weights. The size of the step taken in this direction is controlled using the *FeedfowardFilterStepSize* and *FeedbackFilterStepSize* parameters.
3. The NC-LMS is a modified version of the LMS algorithm which applies only when transpose mode is selected. Note that the NC-LMS has different behavior from the LMS.
4. Applying normalization to the step size causes the filter to vary its step size based on the energy in the filter state vector. This can be used to stabilize convergence in conditions where the strength of the input signal varies with time. When normalization is applied the LMS and NC-LMS become the NLMS and NC-NLMS algorithms respectively.
5. For a detailed description of the LMS family of algorithms see *About Adaptive Equalizer Parts* (algorithm).
6. See also *AdptFltLMS* (algorithm), *AdptFltLMS_Cx* (algorithm), *AdptFltCoreLMS* (algorithm).

Note that since this is a core part, the user must generate the error signal externally and feed it back into the part. This means that the weight update is done on the time step after the output is calculated.

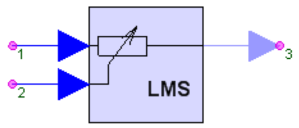
AdptFltCoreLMS Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AdptFltCoreLMS (algorithm)	LMS Adaptive Filter Core

AdptFltCoreLMS



Description: LMS Adaptive Filter Core
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltCoreLMS Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedfowardFilterStepSize	Step size of the algorithm for the EqLib coefficients.	0.4		Float	NO	(0:1)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
FeedbackFilterStepSize	Step size of the algorithm for the feedback stage.	0.4		Float	NO	(0:1)
NLMS	Normalisation of the step size by the input signal power. If on the algorithm used is called NLMS.: OFF , ON	ON		Enumeration	NO	(0:1)
TransposeStructure	Transpose filter structure in the filter part of the adaptive structure.: OFF , ON	OFF		Enumeration	NO	(0:1)
Symmetric	Symmetric feedforward coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoef	Initialized feed forward filter coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoefValue	Sets the feed forward filter coefficients to an inital value (used when InitCoef = YES).	[0.0]		Floating point array	NO	[-∞:∞]

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	real	NO
2	errorSignal	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	real	NO

Notes/Equations

1. This model implements the core of a real arithmetic adaptive filter (FIR or IIR) using the Least Mean Squares (LMS) or Non-Canonical Least Mean Squares (NC-LMS) algorithm. These algorithms try to minimize the mean square of the error signal by adjusting their internal filter coefficients.
2. During each iteration the LMS algorithm approximates the best direction in which to adjust the filter weights based on the current state vector and the current sample of the error signal. This estimate is then used to update the filter weights. The size of the step taken in this direction is controlled using the *FeedfowardFilterStepSize* and *FeedbackFilterStepSize* parameters.
3. The NC-LMS is a modified version of the LMS algorithm which applies only when transpose mode is selected. Note that the NC-LMS has different behavior from the LMS.
4. Applying normalization to the step size causes the filter to vary its step size based on the energy in the filter state vector. This can be used to stabilize convergence in conditions where the strength of the input signal varies with time. When normalization is applied the LMS and NC-LMS become the NLMS and NC-NLMS algorithms respectively.
5. For a detailed description of the LMS family of algorithms see *About Adaptive Equalizer Parts* (algorithm)
6. See also *AdptFltLMS* (algorithm), *AdptFltLMS_Cx* (algorithm), *AdptFltCoreLMS_Cx* (algorithm).

Note that since this is a core part, the user must generate the error signal externally and feed it back into the part. This means that the weight update is done on the time step after the output is calculated.

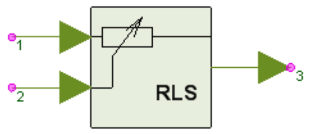
AdptFltCoreRLS_Cx Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AdptFltCoreRLS_Cx (algorithm)	Complex RLS Adaptive Filter Core

AdptFltCoreRLS_Cx



Description: Complex RLS Adaptive Filter Core
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltCoreRLS Cx Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
Lambda	Controls the memory of the algorithm. The larger the value the more memory it has.	0.999		Float	NO	(0:1)
Bias	Bias factor of the RLS algorithm.	1E-05		Float	NO	(-∞:∞)
InitCoef	Initialized feed forward filter coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoefValue	Sets the feed forward filter coefficients to an inital value (used when InitCoef = YES).	[0.0+j*0.0]		Complex array	NO	[-∞:∞]

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	complex	NO
2	errorSignal	complex	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	complex	NO

Notes/Equations

1. This model implements the core of a complex arithmetic adaptive filter (FIR or IIR) using the Recursive Least Squares (RLS) algorithm. The RLS core algorithm tries to minimize the error signal based on the Least Squares (LS) criterion.
2. During each iteration, the RLS algorithm calculates the optimal set of filter weights based on the current state vector, all past state vectors and the corresponding samples of the error signal. The dependence of the result on previous data can be controlled by the parameter *Lambda*. The RLS involves storing and recursively updating the inverse of the data matrix. The diagonal elements of this matrix are initialized to the value of the *Bias* parameter to avoid it becoming singular.
3. For a detailed description of the RLS algorithm see *About Adaptive Equalizer Parts* (algorithm).
4. See also *AdptFltRLS* (algorithm), *AdptFltRLS_Cx* (algorithm), *AdptFltCoreRLS* (algorithm).

Note that since this is a core part, the user must generate the error signal externally and feed it back into the part. This means that the weight update is done on the time step after the output is calculated.

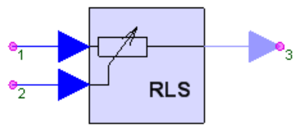
AdptFltCoreRLS Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AdptFltCoreRLS (algorithm)	RLS Adaptive Filter Core

AdptFltCoreRLS



Description: RLS Adaptive Filter Core
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltCoreRLS Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
Lambda	Controls the memory of the algorithm. The larger the value the more memory it has.	0.999		Float	NO	(0:1)
Bias	Bias factor of the RLS algorithm.	1E-05		Float	NO	(-∞:∞)
InitCoef	Initialized feed forward filter coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoefValue	Sets the feed forward filter coefficients to an inital value (used when InitCoef = YES).	[0.0]		Floating point array	NO	[-∞:∞]

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	real	NO
2	errorSignal	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	real	NO

Notes/Equations

1. This model implements the core of a real arithmetic adaptive filter (FIR or IIR) using the Recursive Least Squares (RLS) algorithm. The RLS core algorithm tries to minimize the error signal based on the Least Squares (LS) criterion.
2. During each iteration, the RLS algorithm calculates the optimal set of filter weights based on the current state vector, all past state vectors and the corresponding samples of the error signal. The dependence of the result on previous data can be controlled by the parameter *Lambda*. The RLS involves storing and recursively updating the inverse of the data matrix. The diagonal elements of this matrix are initialized to the value of the *Bias* parameter to avoid it becoming singular.
3. For a detailed description of the RLS algorithm see *About Adaptive Equalizer Parts* (algorithm).
4. See also *AdptFltRLS* (algorithm), *AdptFltRLS_Cx* (algorithm), *AdptFltCoreRLS_Cx* (algorithm).

   Note that since this is a core part, the user must generate the error signal externally and feed it back into the part. This means that the weight update is done on the time step after the output is calculated.

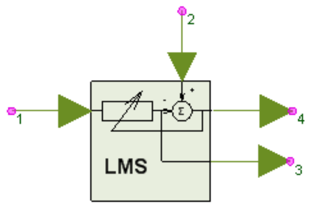
AdptFltLMS_Cx Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AdptFltLMS_Cx (algorithm)	Complex LMS Adaptive Filter

AdptFltLMS_Cx



Description: Complex LMS Adaptive Filter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltLMS Cx Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedfowardFilterStepSize	Step size of the algorithm for the EqLib coefficients.	0.4		Float	NO	(0:1)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
FeedbackFilterStepSize	Step size of the algorithm for the feedback stage.	0.4		Float	NO	(0:1)
NLMS	Normalisation of the step size by the input signal power. If on the algorithm used is called NLMS.: OFF , ON	ON		Enumeration	NO	(0:1)
TransposeStructure	Transpose filter structure in the filter part of the adaptive structure.: OFF , ON	OFF		Enumeration	NO	(0:1)
Symmetric	Symmetric feedforward coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoef	Initialized feed forward filter coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoefValue	Sets the feed forward filter coefficients to an inital value (used when InitCoef = YES).	[0.0+j*0.0]		Complex array	NO	[-∞:∞]

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	complex	NO
2	desiredSignal	complex	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	complex	NO
4	errorSignal	complex	NO

Notes/Equations

1. This model implements a complex arithmetic adaptive filter (FIR or IIR) using the Least Mean Squares (LMS) or Non-Canonical Least Mean Squares (NC-LMS) algorithm. These algorithms try to minimize the Mean Square Error (MSE) between the desired signal and the filter output by adjusting their internal coefficients.
2. During each iteration the LMS algorithm approximates the best direction in which to adjust the filter weights based on the current state vector and the current sample of the desired signal. This estimate is then used to update the filter weights. The size of the step taken in this direction is controlled using the *FeedfowardFilterStepSize* and *FeedbackFilterStepSize* parameters.
3. The NC-LMS is a modified version of the LMS algorithm which applies only when transpose mode is selected. Note that the NC-LMS has different behavior from the LMS.
4. Applying normalization to the step size causes the filter to vary its step size based on the energy in the filter state vector. This can be used to stabilize convergence in conditions where the strength of the input signal varies with time. When normalization is applied the LMS and NC-LMS become the NLMS and NC-NLMS algorithms respectively.
5. For a detailed description of the LMS family of algorithms see *About Adaptive Equalizer Parts* (algorithm)
6. See also *AdptFltLMS* (algorithm), *AdptFltCoreLMS* (algorithm), *AdptFltLMS_Cx* (algorithm).

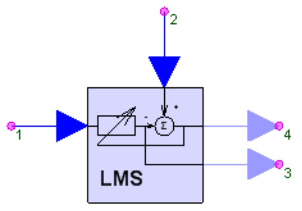
AdptFltLMS Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AdptFltLMS (algorithm)	LMS Adaptive Filter

AdptFltLMS



Description: LMS Adaptive Filter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltLMS Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedfowardFilterStepSize	Step size of the algorithm for the EqLib coefficients.	0.4		Float	NO	(0:1)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
FeedbackFilterStepSize	Step size of the algorithm for the feedback stage.	0.4		Float	NO	(0:1)
NLMS	Normalisation of the step size by the input signal power. If on the algorithm used is called NLMS.: OFF , ON	ON		Enumeration	NO	(0:1)
TransposeStructure	Transpose filter structure in the filter part of the adaptive structure.: OFF , ON	OFF		Enumeration	NO	(0:1)
Symmetric	Symmetric feedforward coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoef	Initialized feed forward filter coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoefValue	Sets the feed forward filter coefficients to an inital value (used when InitCoef = YES).	[0.0]		Floating point array	NO	[-∞:∞]

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	real	NO
2	desiredSignal	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	real	NO
4	errorSignal	real	NO

Notes/Equations

1. This model implements a real arithmetic adaptive filter (FIR or IIR) using the Least Mean Squares (LMS) or Non-Canonical Least Mean Squares (NC-LMS) algorithm. These algorithms try to minimize the Mean Square Error (MSE) between the desired signal and the filter output by adjusting their internal coefficients.
2. During each iteration the LMS algorithm approximates the best direction in which to adjust the filter weights based on the current state vector and the current sample of the desired signal. This estimate is then used to update the filter weights. The size of the step taken in this direction is controlled using the *FeedfowardFilterStepSize* and *FeedbackFilterStepSize* parameters.
3. The NC-LMS is a modified version of the LMS algorithm which applies only when transpose mode is selected. Note that the NC-LMS has different behavior from the LMS.
4. Applying normalization to the step size causes the filter to vary its step size based on the energy in the filter state vector. This can be used to stabilize convergence in conditions where the strength of the input signal varies with time. When normalization is applied the LMS and NC-LMS become the NLMS and NC-NLMS algorithms respectively.
5. For a detailed description of the LMS family of algorithms see *About Adaptive Equalizer Parts* (algorithm)
6. See also *AdptFltLMS_Cx* (algorithm), *AdptFltCoreLMS* (algorithm), *AdptFltCoreLMS_Cx* (algorithm)

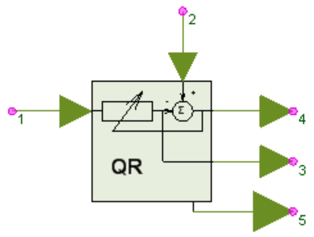
AdptFltQR_Cx Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AdptFltQR_Cx (algorithm)	Complex QR Adaptive Filter

AdptFltQR_Cx



Description: Complex QR Adaptive Filter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltQR Cx Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
Lambda	Controls the memory of the algorithm. The larger the value the more memory it has.	0.999		Float	NO	(0:1)

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	complex	NO
2	desiredSignal	complex	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	complex	NO
4	errorSignal	complex	NO
5	posterioriError	complex	NO

Notes/Equations

1. This model implements a complex arithmetic adaptive filter (FIR or IIR) using the QR Recursive Least Squares (QR-RLS) algorithm. The QR-RLS algorithm tries to minimize the error between the desired signal and the output of the filter based on the Least Squares (LS) criterion.
2. During each iteration, the QR-RLS algorithm calculates the optimal set of filter weights based on the current state vector, all past state vectors and the corresponding samples of the desired signal. The dependence of the result on previous data can be controlled by the parameter *Lambda*. Internally the QR-RLS uses a QR decomposition of the data matrix to store its current state.
3. For a detailed description of the QR-RLS algorithm see *About Adaptive Equalizer Parts* (algorithm).
4. See also *AdptFltQR* (algorithm).

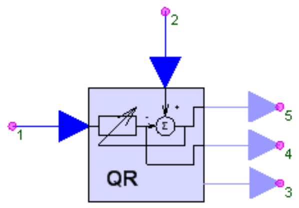
AdptFltQR Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AdptFltQR (algorithm)	QR Adaptive Filter

AdptFltQR



Description: QR Adaptive Filter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltQR Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
Lambda	Controls the memory of the algorithm. The larger the value the more memory it has.	0.999		Float	NO	(0:1)

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	real	NO
2	desiredSignal	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	real	NO
4	errorSignal	real	NO
5	posterioriError	real	NO

Notes/Equations

1. This model implements a real arithmetic adaptive filter (FIR or IIR) using the QR Recursive Least Squares (QR-RLS) algorithm. The QR-RLS algorithm tries to minimize the error between the desired signal and the output of the filter based on the Least Squares (LS) criterion.
2. During each iteration, the QR-RLS algorithm calculates the optimal set of filter weights based on the current state vector, all past state vectors and the corresponding samples of the desired signal. The dependence of the result on previous data can be controlled by the parameter *Lambda*. Internally the QR-RLS uses a QR decomposition of the data matrix to store its current state.
3. For a detailed description of the QR-RLS algorithm see *About Adaptive Equalizer Parts* (algorithm).
4. See also *AdptFltQR_Cx* (algorithm).

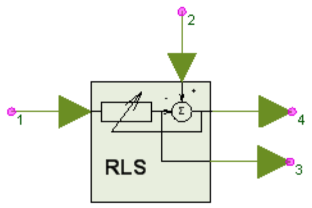
AdptFltRLS_Cx Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AdptFltRLS_Cx (algorithm)	Complex RLS Adaptive Filter

AdptFltRLS_Cx



Description: Complex RLS Adaptive Filter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltRLS Cx Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
Lambda	Controls the memory of the algorithm. The larger the value the more memory it has.	0.999		Float	NO	(0:1)
Bias	Bias factor of the RLS algorithm.	1E-05		Float	NO	(-∞:∞)
InitCoef	Initialized feed forward filter coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoefValue	Sets the feed forward filter coefficients to an inital value (used when InitCoef = YES).	[0.0+j*0.0]		Complex array	NO	[-∞:∞]

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	complex	NO
2	desiredSignal	complex	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	complex	NO
4	errorSignal	complex	NO

Notes/Equations

1. This model implements a complex arithmetic adaptive filter (FIR or IIR) using the Recursive Least Squares (RLS) algorithm. The RLS algorithm tries to minimize the error between the desired signal and the output of the filter based on the Least Squares (LS) criterion.
2. During each iteration, the RLS algorithm calculates the optimal set of filter weights based on the current state vector, all passed input vectors and the corresponding samples of the desired signal. The dependence of the result on previous data can be controlled by the parameter *Lambda*. The RLS involves storing and recursively updating the inverse of the data matrix. The diagonal elements of this matrix are initialized to the value of the *Bias* parameter to avoid it becoming singular.
3. For a detailed description of the RLS algorithm see *About Adaptive Equalizer Parts* (algorithm).
4. See also *AdptFltRLS* (algorithm), *AdptFltCoreRLS* (algorithm), *AdptFltCoreRLS_Cx* (algorithm).

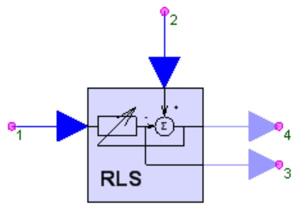
AdptFltRLS Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AdptFltRLS (algorithm)	RLS Adaptive Filter

AdptFltRLS



Description: RLS Adaptive Filter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdptFltRLS Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FeedforwardFilterLength	Number of filter coefficients of the EqLib filter stage.	10		Integer	NO	(1:∞)
FeedbackFilterLength	Number of filter coefficients of the feedback filter stage. A value different from zero makes the structure an IIR setup.	0		Integer	NO	(0:∞)
Lambda	Controls the memory of the algorithm. The larger the value the more memory it has.	0.999		Float	NO	(0:1)
Bias	Bias factor of the RLS algorithm.	1E-05		Float	NO	(-∞:∞)
InitCoef	Initialized feed forward filter coefficients: OFF , ON	OFF		Enumeration	NO	(0:1)
InitCoefValue	Sets the feed forward filter coefficients to an inital value (used when InitCoef = YES).	[0.0]		Floating point array	NO	[-∞:∞]

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	real	NO
2	desiredSignal	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	outputSignal	real	NO
4	errorSignal	real	NO

Notes/Equations

1. This model implements a real arithmetic adaptive filter (FIR or IIR) using the Recursive Least Squares (RLS) algorithm. The RLS algorithm tries to minimize the error between the desired signal and the output of the filter based on the Least Squares (LS) criterion.
2. During each iteration, the RLS algorithm calculates the optimal set of filter weights based on the current state vector, all passed state vectors and the corresponding samples of the desired signal. The dependence of the result on previous data can be controlled by the parameter *Lambda*. The RLS involves storing and recursively updating the inverse of the data matrix. The diagonal elements of this matrix are initialized to the value of the *Bias* parameter to avoid it becoming singular.
3. For a detailed description of the RLS algorithm see *About Adaptive Equalizer Parts* (algorithm).
4. See also *AdptFltRLS_Cx* (algorithm), *AdptFltCoreRLS* (algorithm), *AdptFltCoreRLS_Cx* (algorithm).

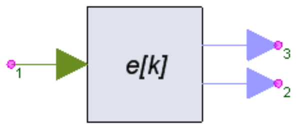
ErrorFilterCx Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
ErrorFilterCx (algorithm)	Complex Error Filter

ErrorFilterCx



Description: Complex Error Filter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *ErrorFilterCx Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FloorType	Specifies whether the floor parameter introduced has to be interpreted as a linear or as logarithmic value.: Linear , dB	Linear		Enumeration	NO	(0:1)
Floor	Minimum value representable at the logarithmic output of this component. This value avoids the calculation of the logarithm of zero.	-200		Float	NO	(-∞:∞)
FilterLength	Length of the Gaussian filter. If set to one no filtering is performed.	10		Integer	NO	(1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	complex	NO

Output Ports

Port	Name	Signal Type	Optional
2	outputLinear	real	NO
3	outputLog	real	NO

Notes/Equations

1. This model measures the instantaneous power of its complex valued input signal by computing its magnitude squared and applying a Gaussian filter to smooth the result. It is provided as a convenient way to smooth the appearance of an error signal such as that produced by an adaptive filtering algorithm. However it can be used to measure the power of any signal.
2. The length of the Gaussian filter can be controlled by the *FilterLength* parameter. The bandwidth of the filter is automatically varied according to the length. Choose *FilterLength*=1 to disable filtering.
3. The resulting signal is provided in both linear and logarithmic formats as two separate outputs. The logarithmic output is $10*\log_{10}(g(k)+F)$ where $g(k)$ is the output of the Gaussian filter. If *FloorType* is *Linear* then *F*=*Floor*. If *FloorType* is *dB* then $F=10^{(Floor\ /\ 10)}$.
4. See also *ErrorFilter* (algorithm).

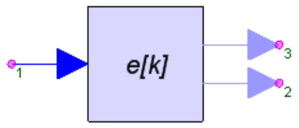
ErrorFilter Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
ErrorFilter (algorithm)	Error Filter

ErrorFilter



Description: Error Filter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *ErrorFilter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FloorType	Specifies whether the floor parameter introduced has to be interpreted as a linear or as logarithmic value.: Linear , dB	Linear		Enumeration	NO	(0:1)
Floor	Minimum value representable at the logarithmic output of this component. This value avoids the calculation of the logarithm of zero.	-200		Float	NO	(-∞:∞)
FilterLength	Length of the Gaussian filter. If set to one no filtering is performed.	10		Integer	NO	(1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	outputLinear	real	NO
3	outputLog	real	NO

Notes/Equations

1. This model measures the instantaneous power of its real valued input signal by squaring it and applying a Gaussian filter to smooth the result. It is provided as a convenient way to smooth the appearance of an error signal such that produced by an adaptive filtering algorithm. However it can be used to measure the power of any signal.
2. The length of the Gaussian filter can be controlled by the *FilterLength* parameter. The bandwidth of the filter is automatically varied according to the length. Choose *FilterLength*=1 to disable filtering.
3. The resulting signal is provided in both linear and logarithmic formats as two separate outputs. The logarithmic output is $10*\log_{10}(g(k)+F)$ where $g(k)$ is the output of the Gaussian filter. If *FloorType* is *Linear* then *F*=*Floor*. If *FloorType* is *dB* then $F=10^{(Floor\ /\ 10)}$.
4. See also *ErrorFilterCx* (algorithm).

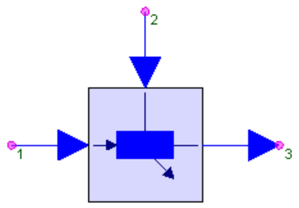
LMS Part

Categories: *Adaptive Equalizers* (algorithm), *C++ Code Generation* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
LMS (algorithm)	LMS Adaptive Filter
LMS_Cx (algorithm)	Complex LMS Adaptive Filter

LMS (LMS Adaptive Filter)



Description: LMS Adaptive Filter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *LMS Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Taps	Filter tap values	[-0.040609, -0.001628, 0.17853, 0.37665, 0.37665, 0.17853, -0.001628, -0.040609]		Floating point array	NO
m_iDecimation	Decimation ratio	1		Integer	NO
DecimationPhase	Decimation phase	0		Integer	NO
StepSize	Adaptation step size	0.01		Float	YES
ErrorDelay	Update loop delay	1		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO
2	error	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	output	real	NO

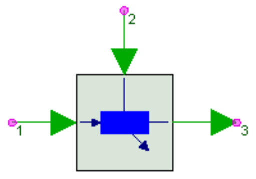
Notes/Equations

1. LMS is an adaptive filter using the least-mean square algorithm. The initial filter coefficients are given by the Taps parameter. The default initial coefficients give an 8th-order, linear phase lowpass filter. LMS supports decimation, but not interpolation.
2. When used correctly, this LMS adaptive filter will try to minimize the mean-squared error of the signal at its error input [1]. The output of the filter should be compared to (subtracted from) some reference signal to produce an error signal. That error signal should be fed back to the error input. The ErrorDelay parameter must equal the total number of delays in the path from the output of the filter back to the error input. This ensures correct alignment of the adaptation algorithm. The number of delays must be greater than 0 or the simulation will deadlock.
   The adaptation algorithm is the well-known LMS, or stochastic-gradient, algorithm.
3. If the SaveTapsFile parameter is not empty, a file will be created with the name given by this parameter, and the final tap values will be stored there at the end of the simulation.
4. See also: *LMS_Cx* (algorithm)

References

1. S. Haykin, *Adaptive Filter Theory*, Prentice Hall: Englewood Cliffs, NJ. 1991. 2nd ed.

LMS_Cx (Complex LMS Adaptive Filter)



Description: Complex LMS Adaptive Filter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *LMS Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Taps	Filter tap values	[-0.040609, -0.001628, 0.17853, 0.37665, 0.37665, 0.17853, -0.001628, -0.040609]		Complex array	NO
Decimation	Decimation ratio	1		Integer	NO
DecimationPhase	Decimation phase	0		Integer	NO
StepSize	Adaptation step size	0.01		Float	YES
ErrorDelay	Update loop delay	1		Integer	NO
SaveTapsFile	Filename in which to save final tap values			Text	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	complex	NO
2	error	complex	NO

Output Ports

Port	Name	Signal Type	Optional
3	output	complex	NO

Notes/Equations

1. LMS_Cx is a complex adaptive filter using the least-mean square algorithm. For more details see *LMS* (algorithm).

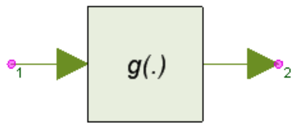
NonLinearityCx Part

Categories: *Adaptive Equalizers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
NonLinearityCx (algorithm)	Non-linear function for use in blind adaptive algorithms.

NonLinearityCx



Description: Non-linear function for use in blind adaptive algorithms.
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *NonLinearityCx Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
BlindAlgorithm	Choice of blind adaptive algorithm to use.: Decision Directed (Luckly) , Constant Modulus Algorithm (Godard) , Sato , Benveniste-Goursat , Stop-and-Go (Picchi and Prati)	Decision Directed (Luckly)		Enumeration	NO	(0:4)
ModType	Modulation type: BPSK , QPSK , PSK8 , PSK16 , QAM16 , QAM32 , QAM64 , QAM128 , 256QAM , User_Defined	BPSK		Enumeration	NO	(0:9)
MappingTable	Constellation table (used when ModType=User_Defined)	[1+j*0,-1+j*0]		Complex array	NO	[-∞:∞]

Input Ports

Port	Name	Signal Type	Optional
1	inputSignal	complex	NO

Output Ports

Port	Name	Signal Type	Optional
2	outputSignal	complex	NO

Notes/Equations

1. This model implements a set of non-linear functions for use in blind adaptive algorithms. A range of blind adaptive algorithms can be constructed by using this part in conjunction with the adaptive filter cores.
2. The DDM and Godard algorithms work with complex-valued or real-valued modulation formats. Sato, Benveniste-Goursat and Stop-and-Go algorithms only work with real-valued modulation formats.

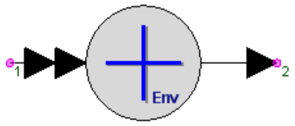
AddEnv Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AddEnv (algorithm)	Envelope Signal Adder

AddEnv (Complex Envelope Signal Adder)



Description: Envelope Signal Adder
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Add Part* (algorithm), *AddEnv Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
OutputFc	Output characterization frequency for the combined signal: Min, Max, Center, User defined	Center		Enumeration	NO		O
UserDefinedFc	User defined output characterization frequency	100e6	Hz	Float	NO	[0:∞)†	F

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signals	multiple envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	envelope	NO

Notes/Equations

1. The AddEnv model outputs the sum of the complex envelope inputs.
2. This model reads 1 sample from all inputs and writes 1 sample to the output.
3. Each input may have either a real baseband value or a complex value defined at a specific characterization frequency.
4. The output characterization frequency NewFc is determined by the configuration of inputs and the parameters, OutputFc and UserDefinedFc.
   - If any input is a real baseband signal, then the output will be a real baseband signal.
   - Otherwise, the output will be a complex envelope signal defined at a characterization frequency NewFc as follows.
     - If OutputFc is ***Min***, NewFc is set to the minimum input characterization frequencies.
     - If OutputFc is ***Max***, NewFc is set to the maximum input characterization frequencies.
     - If OutputFc is ***Center***, NewFc is set to the average of the maximum and minimum input characterization frequencies.
     - If OutputFc is ***User defined***, NewFc is set to the UserDefinedFc parameter.

     Note that if UserDefinedFc is positive, then all inputs require a nonzero characterization frequency.
5. All inputs are converted to their equivalent representation at NewFc before summing.
6. A complex envelope value OldIQ at characterization frequency OldFc is converted to a NewIQ at characterization frequency NewFc at time t as follows.
   - NewIQ is set to $OldIQ \cdot \exp(j2\pi(OldFc - NewFc)t)$
   - If NewFc is zero, the imaginary part of NewIQ is set to zero.
7. A warning message is displayed once if an input sample rate is too small to make the transformation without loss of information.

See:
SubEnv (algorithm)
MpyEnv (algorithm)
MpyMultiEnv (algorithm)

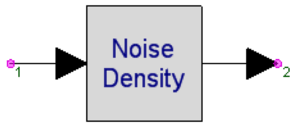
AddNDensity Part

Categories: *Analog/RF* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AddNDensity (algorithm)	Add Noise Density to Input

AddNDensity (Add Noise Density)



Description: Add Noise Density to Input
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *AddNDensity Part* (algorithm)

Model Parameters

Name	**Description**	**Default**	**Units**	**Type**	**Runtime Tunable**	**Range**
NDensityType	Noise density type: Constant noise density, Noise density vs freq	Constant noise density		Enumeration	NO	
NDensity	Noise power spectral density	0.0	W	Float	NO	[0:∞)
NDensityFreq	Noise spectral density specification frequency values (Hz)			Floating point array	NO	
NDensityPower	Noise spectral density specification power density values (dBm/Hz)			Floating point array	NO	
RefR	Reference resistance	50	ohm	Float	NO	(0:∞)

Input Ports

Port	**Name**	**Description**	**Signal Type**	**Optional**
1	input	input signal	envelope	NO

Output Ports

Port	**Name**	**Description**	**Signal Type**	**Optional**
2	output	output signal	envelope	NO

Notes/Equations

1. This model adds noise to the input signal.
2. At every execution, it reads 1 sample from the input and writes 1 sample to the output.
3. If *NDensityType* is set to *Constant noise density*, then the noise added is white Gaussian.
   The noise density is specified in the *NDensity* parameter. Although the units for this parameter are power units the value is interpreted as power spectral density, that is, power per frequency unit (Hz).
   The total noise power added to the input signal is *NDensity* × *BW*, where
   - *NDensity* is the noise power spectral density in Watts/Hz
   - *BW* is the simulation bandwidth (equal to *SR* for a complex envelope signal and *SR / 2* for a real baseband signal; *SR* is the input signal sample rate) in Hz.
     The rms noise voltage level is $\sqrt{NDensity \cdot BW \cdot RefR}$
     .
     The value of *NDensity* (in Watts/Hz) is related to temperature (in °Kelvin) as *k*·*T* , where *k* is the Boltzmann constant (1.3806504e-23). At the standard system temperature of 290° Kelvin (16.85° Celsius), the *NDensity* is 4.00388587e-21 Watts/Hz (-173.975 dBm/Hz).
4. If *NDensityType* is set to *Noise density vs freq*, then the spectral profile of the noise added can be specified using the NDensityFreq (values need to be in Hz) and NDensityPower (values need to be in dBm/Hz) array parameters.

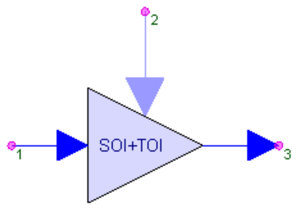
AmplifierBB Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AmplifierBB (algorithm)	Baseband Polynomial Amplifier with Noise Figure

AmplifierBB (Baseband Nonlinearity With Noise Figure)



Description: Baseband Polynomial Amplifier with Noise Figure
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *AmplifierBB Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Offset	DC offset voltage	0		Float	NO	(-∞:∞)
GainUnit	Gain unit for the Gain parameter or the optional voltage controlled gain input: voltage, dB	voltage		Enumeration	NO	
Gain	Gain with units defined by GainUnit (used if optional voltage controlled gain input not used)	1		Float	NO	(-∞:∞)
NoiseFigure	Input noise figure in dB	0		Float	NO	[0:∞)
SOIout	Output second order intercept power in dBm	40		Float	NO	(-∞:∞)
TOIout	Output third order intercept power in dBm	30		Float	NO	(-∞:∞)
HigherOrderTerms	Higher order intercept terms 4th to 11th in dBm	[ ]		Floating point array	NO	(-∞:∞)
RefR	Reference resistance	50	ohm	Float	NO	(0:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	real	NO
2	control	gain control	real	YES

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output signal	real	NO

Notes/Equations

1. AmplifierBB models a baseband nonlinearity including noise figure.
2. This block reads 1 sample from input and writes 1 sample to output.
3. The small signal gain value is based on either the *Gain* parameter value or on the optional control value. The control input is optional and when used the value at that pin is used instead of the *Gain* parameter value. In the following discussion, the *Gain* value generally refers to the control input value as well.
4. The nonlinearity is defined by the parameters *Gain*, *SOIout*, *TOIout* and *RefR* and by curve fitting the nonlinear parameters with a second, third, or higher order polynomial. The higher order polynomial is used if the *HigherOrderTerms* are specified. SOI, TOI and HigherOrderTerms are ignored if set to 200 dBm or higher.
5. Nonlinear model with SOI is defined as follows.
   - Let $y = c_1 x - c_2 x^2$
     where
     - x = input voltage
     - y = output voltage
     - c_1 = small signal gain = *Gain*

    - c $_2$ = second order gain factor
  - Saturation occurs at the peak x = x_max. This occurs when dy/dx = 0
    - c $_1$ - 2 c $_2$ x_max = 0. Therefore, x_max = (c $_1$ / 2) / c $_2$.
    - y_max = c $_1$ x_max - c $_2$ x_max ^{2}
  - Given the output, *SOI* is specified in dBm into *RefR* ohms, then c $_2$ is derived as follows.
    - Let SOIv be *SOI* in terms of output peak volts. $SOIv = \sqrt{2 \cdot RefR \cdot 10^{\frac{SOI-30}{10}}}$
    - *SOI* occurs at x $_i$ such that c $_1$ x $_i$ = c $_2$ x $_i$ ^{2} = SOIv. Therefore, c $_2$ = c $_1$ ^{2} / SOIv.
      This value for c $_2$ implies that a positive signal experiences expansion, but a negative signal will have compression.

6. Nonlinear model with *TOI*
  - Let y = c $_1$ x + c $_3$ x ^{3}
    where
    - x = input voltage
    - y = output voltage
    - c $_1$ = *Gain*
    - c $_3$ = third order gain factor
  - Saturation occurs at the peak x = x_max. This occurs when dy/dx = 0
    - c $_1$ + 3 c $_3$ x_max ^{2} = 0
    - $x\_max = \sqrt{\frac{-c_1}{3} \cdot \frac{1}{c_3}}$
    - y_max = c $_1$ x_max + c $_3$ x_max ^{3}
  - Given the output, *TOI* is specified in dBm into *RefR* ohms, then c $_3$ is derived as follows.
    - Let TOIv be *TOI* in terms of output peak volts. $TOIv = \sqrt{2 \cdot RefR \cdot 10^{\frac{TOI-30}{10}}}$
    - *TOI* is defined for a two tone input signal, x(t). $x(t) = a_1 \cos(w_1 t) + a_2 \cos(w_2 t)$
    - The expansion for y(t) = c $_1$ x(t) + c $_3$ x ^{3}(t) results in fundamental terms at w1, w2 and third order terms at 2 w1 - w2, 2 * w2 - w1.
      See note One and Two Tone Response for SOI and TOI.
      After the trigonometric expansion,
      - the first order term is c $_1$
      - the third order term is: (3/4) c $_3$ x ^{3}
    - *TOI* occurs at x $_i$ such at c $_1$ x $_i$ = -(3/4) c $_3$ x ^{3} = TOIv
      Thus c $_3$ = -(4/3) c $_1$ ^{3} / TOIv ^{2}

7. Nonlinear model with *SOI* and *TOI*
  - Let y = c $_1$ x + c $_2$ x ^{2} + c $_3$ x ^{3}
    where
    - x = input voltage
    - y = output voltage
    - c $_1$ = *Gain*
    - c $_2$ = second order gain factor
    - c $_3$ = third order gain factor
    - c $_2$ and c $_3$ are derived as described in the notes above.
  - Saturation occurs at the peak x = x_max. This occurs when dy/dx = 0
    - c $_1$ - 2 c $_2$ x_max + 3 c $_3$ x_max ^{2} = 0
    - Using the quadratic equations solution that makes x_max positive
      $x\_max = \frac{-1}{2 \cdot 3 \cdot c_3}\left(-2c_2 + \sqrt{(-2c_2)^2 - 4c_1 \cdot 3c_3}\right)$

      This value of x_max is always positive.
  - y_max = c $_1$ x_max - c $_2$ x_max ^{2} + c $_3$ x_max ^{3}
    This y_max value is the limit for a positive x_max signal.
    For a negative signal, there will be another limiting value that is calculated using the above expression for the other quadratic equation solution.

8. When HigherOrderTerms are defined (with values less than 200 dBm), then a similar process to that described above is used to derive the nonlinearity polynomial coefficiants along with the positive and negative saturation characteristics.
9. The noise is defined by the parameters *NoiseFigure* and *RefR*.
  - Let
    - k = Boltzmann constant = 1.3806504e-23 Joules/Kelvin
    - sr = simulation sample rate at this block input
    - t0 = reference temperature at 290 Kelvin
    - nf = 10 $^{(NoiseFigure/10)}$
  - Then, the rms noise voltage, vn_rms is calculated as follows.

$$vn\_rms = \sqrt{k \cdot (t0 \cdot (nf - 1)) \cdot \frac{sr}{2} \cdot RefR}$$

vn_rms is used to generate a Gaussian random variate that is added to the input signal before the signal is amplified through the nonlinearity.

One and Two-Tone Response for SOI and TOI

-
    - Let $y = c_1 x + c_2 x^2 + c_3 x^3$
      where
        - y = output voltage
        - c_1 = small signal gain
        - c_2 = second order gain factor
        - c_3 = third order gain factor
    - For one-tone excitation, $x(t) = A_1 \cos(\omega_1 t)$

      , the response is:

      $$y(t) = \frac{a_2}{2} A_1^2 + \left(a_1 A_1 + \frac{3a_3}{4} A_1^3\right) \cos(\omega_1 t) + \left(\frac{a_2}{2} A_1^2\right) \cos(2\omega_1 t) + \left(\frac{a_3}{4} A_1^3\right) \cos(3\omega_1 t)$$
    - For two-tone excitation, $x(t) = A_1 \cos(\omega_1 t) + A_2 \cos(\omega_2 t)$

      , the response is:

$$\begin{aligned}
y(t) = &\frac{1}{2} a_2 (A_1^2 + A_2^2) \\
&+ \left(a_1 A_1 + \frac{3}{4} a_3 A_1^3 + \frac{3}{2} a_3 A_1 A_2^2\right) \cos(\omega_2 t) \\
&+ \left(a_1 A_2 + \frac{3}{4} a_3 A_2^3 + \frac{3}{2} a_3 A_2 A_1^2\right) \cos(\omega_2 t) \\
&+ \frac{1}{2} a_2 A_1^2 \cos(2\omega_1 t) \\
&+ \frac{1}{2} a_2 A_2^2 \cos(2\omega_2 t) \\
&+ \frac{1}{4} a_3 A_1^3 \cos(3\omega_1 t) \\
&+ \frac{1}{4} a_3 A_2^3 \cos(3\omega_2 t) \\
&+ a_2 A_1 A_2 \cos\left((\omega_1 - \omega_2) t\right) \\
&+ a_2 A_1 A_2 \cos\left((\omega_1 + \omega_2) t\right) \\
&+ \frac{3}{4} a_3 A_2 A_1^2 \cos\left((2\omega_1 + \omega_2) t\right) \\
&+ \frac{3}{4} a_3 A_2 A_1^2 \cos\left((2\omega_1 - \omega_2) t\right) \\
&+ \frac{3}{4} a_3 A_1 A_2^2 \cos\left((2\omega_2 + \omega_1) t\right) \\
&+ \frac{3}{4} a_3 A_1 A_2^2 \cos\left((2\omega_2 - \omega_1) t\right)
\end{aligned}$$

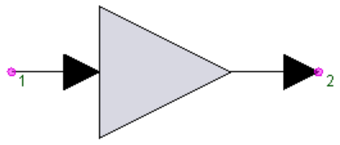
Amplifier Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Amplifier (algorithm)	Nonlinear Amplifier with Noise Figure

Amplifier (Nonlinear Gain)



Description: Nonlinear Amplifier with Noise Figure
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Gain Part* (algorithm), *Amplifier Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
GainUnit	Gain unit for the Gain parameter or the optional voltage controlled gain input: voltage, dB	voltage		Enumeration	NO	
Gain	Gain with units defined by GainUnit (used if optional voltage controlled gain input not used)	1		Float	NO	(-∞:∞)
NoiseFigure	Input noise figure in dB	0		Float	NO	[0:∞)
GCType	Gain compression type: none, TOI, dBc1, TOI+dBc1, PSat+GCSat+TOI, PSat+GCSat+dBc1, PSat+GCSat+TOI+dBc1, RappNonlinearity, Gain compression vs input power, AM/AM and AM/PM vs input power	none		Enumeration	NO	
TOIout	Output third order intercept power	0.1	W	Float	NO	(0:∞)†
dBc1out	Output 1 dB gain compression power	0.01	W	Float	NO	(0:∞)†
PSat	Saturation power	0.032	W	Float	NO	(0:∞)†
GCSat	Gain compression at saturation in dB	3		Float	NO	[3:7]†
RappS	Rapp nonlinearity smoothness factor	3		Integer	NO	(0:∞)
GComp	Array of triple values for Input Power(dBm) and either Gain(dB)/Phase(deg) change from small signal or AM-to-AM(dB/dB)/AM-to-PM(deg/dB)	[0, 0, 0]		Floating point array	NO	‡
RefR	Reference resistance	50	ohm	Float	NO	(0:∞)

Input Ports

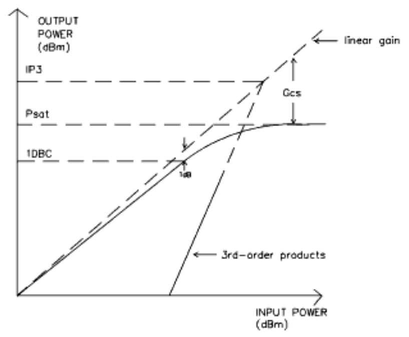
Port	Name	Description	Signal Type	Optional
1	input	input signal	envelope	NO
2	control	gain control	real	YES

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output signal	envelope	NO

Notes/Equations

1. Amplifier models a nonlinearity including noise figure for use with either baseband or complex envelope signals.
2. This block reads 1 sample from input in and writes 1 sample to output out.
3. The small signal gain value is based on either the *Gain* parameter value or on the optional control input value if used. The control input does not need to be connected. It is optional and when used the value at that pin is used instead of the *Gain* parameter value. In the following discussion, the *Gain* value implies the control input value as well.
4. The nonlinearity is defined by the parameters *GCType*, *TOIout*, *dBc1out*, *PSat*, *GCSat*, *RappS*, *GComp* and *RefR*. All nonlinear models except those based on *GComp* are achieved by curve fitting nonlinear parameters with an odd order nonlinear polynomial. The *GComp* based nonlinearities are table based with interpolation between points.
5. The general nonlinearity has this output power versus input power characteristic.



6. Nonlinear model with TOI
   - Let $y = c_1 x + c_3 x^3$
     where
     - x = input voltage
     - y = output voltage
     - c1 = small signal gain (*Gain*)
     - c3 = third order gain factor
   - See detail discussion in the documentation for the AmplifierBB block. See *AmplifierBB* (algorithm).
7. Nonlinear model with dBc1
   - Let $y = c_1 x + c_3 x^3$
     where
     - x = input voltage
     - y = output voltage
     - c1 = small signal gain (*Gain*)
     - c3 = third order gain factor
   - Given the output *dBc1out* in dBm into *RefR* ohms, then c_3 is derived as follows.
     - dBc1v = *dBc1out* in terms of output peak volts
       $$dBc1v = \sqrt{2 \cdot RefR \cdot 10^{\frac{dBc1out-30}{10}}}$$
     - *dBc1out* occurs when y is 1 dB compressed.
   - This occurs when $y_1 = c_1 x_1 + c_3 x_1^3 = 0.891 c_1 x_1$
     where
     - -1 dB = 20 $\log_{10}$ (0.891)
     - y_1 = dBc1v
     - x_1 = (dBc1v / 0.891) / c_1
     - $c_3 = (y_1 - c_1 x_1) / x_1^3$
   - Saturation occurs as for the TOI nonlinearity:
     - $$x\_max = \sqrt{\frac{-c1/3}{c3}}$$
     - $y\_max = c_1 x\_max + c_3 x\_max^3$
8. Nonlinear model with TOI and dBc1
   - Let $y = c_1 x + c_3 x^3 + c_5 x^5$
     where
     - x = input voltage
     - y = output voltage
     - c_1 = small signal gain
     - c_3 = third order gain factor
     - c_5 = fifth order gain factor
     - c_3 is same as that derived for the TOI nonlinearity.
   - Given the output *dBc1out* in dBm into *RefR* ohms, then c_5 is derived as follows:
     - dBc1v = dBc1 in terms of output peak volts
       $$dBc1v = \sqrt{2 \cdot RefR \cdot 10^{\frac{dBc1out-30}{10}}}$$
   - dBc1 occurs when y is 1 dB compressed.
     This occurs when $y_1 = c_1 x_1 + c_3 x_1^3 + c_5 x^5 = 0.891 c_1 x_1$
     where
     - -1 dB = 20 $\log_{10}$ (0.891)
     - y_1 = dBc1v
     - x_1 = (dBc1v / 0.891) / c_1
     - $c_5 = (y_1 - c_1 x_1 - c_3 x_1^3) / x_1^5$
   - The minimum *TOI* occurs when $c_5 = 0$

Therefore, the minimum *TOI* is defined when

- $c_3 = -(1 - 0.891)(0.891^2)(c_1^3) / dBc1v^2$

- Saturation occurs at the peak x = x_max. This occurs when dy/dx = 0
  - $c_1 + 3 c_3 x\_max^2 + 5 c_5 x\_max^4 = 0$
  - Using the minimum quadratic equations solution

    $$x\_max = \sqrt{\frac{1}{2\cdot 5\cdot c_5}\cdot(-3c_3 - \sqrt{(3c_3)^2 - 4c_1\cdot 5c_5}}$$

    $y\_max = c_1 x\_max + c_3 x\_max^3 + c_5 x\_max^5$

9. Nonlinear model with PSat, GCSat, TOI and dBc1
   - Let $y = c_1 x + c_3 x^3 + c_5 x^5 + c_7 x^7 + c_9 x^9 + c_{11} x^{11}$

     where
     - x = input voltage
     - y = output voltage
     - c_1 = small signal gain
     - c_3, c_5, c_7, c_9, c_{11} = 3rd, 5th, 7th, 9th, 11th order gain factor; $c_3 < 0$, $c_5 < 0$
   - For *GCType = PSat+GCSat+TOI*, *dBc1out = TOIout* - dB_offset
     For *GCType = PSat+GCSat+dBc1*, *TOIout = dBc1out* + dB_offset
     For complex envelope signals, dB_offset = 10.6357 dB, otherwise for baseband signals, dB_offset = 11.8851 dB.
   - The c_{11} factor is used to improve the stability of the polynomial expression; otherwise the polynomial is more likely to have gain expansion or reduced range of usage.
   - c_3 is based on *TOIout* only and is as derived for the TOI nonlinearity.
   - GCSatR = voltage compression ratio = $10^{(GSat / 20)}$

     y_s = output peak voltage at saturation = $PSatv = \sqrt{2\cdot RefR\cdot 10^{\frac{PSat-30}{10}}}$
     x_s = input peak voltage at saturation = PSatv × GCSatR / c_1

     y_1 = output peak voltage at 1dB compression = $dBc1v = \sqrt{2\cdot RefR\cdot 10^{\frac{dBc1out-30}{10}}}$
     x_1 = input peak voltage at 1dB compression = (dBc1v / 0.891) / c_1
   - To derive c_5, c_7, c_9 and c_{11}. Setup 4 equations with 4 unknowns.

     At 1dB compression:
     $c_1 x_1 + c_3 x_1^3 + c_5 x_1^5 + c_7 x_1^7 + c_9 x_1^9 + c_{11} x_1^{11} = y_1$
     At saturation there is *GCSat* dB compression
     $c_1 x_s + c_3 x_s^3 + c_5 x_s^5 + c_7 x_s^7 + c_9 x_s^9 + c_{11} x_s^{11} = y_s$
     At saturation set dy/dx = 0
     $c_1 + 3 c_3 x_1^2 + 5 c_5 x_1^4 + 7 c_7 x_1^6 + 9 c_9 x_1^8 + 11 c_{11} x_1^{10} = 0$
     At saturation set the second derivative of y with respect to x to zero
     $2 \times 3 c_3 x_1 + 4 \times 5 c_5 x_1^3 + 6 \times 7 c_7 x_1^5 + 8 \times 9 c_9 x_1^7 + 10 \times 11 c_{11} x_1^9 = 0$
   - The four equations are solved for the four unknowns: c_5, c_7, c_9, c_{11}
   - The following parameter value limits are defined.
     - For *GCType = PSat+GCSat+TOI+dBc1*
       dBc1out_dBm + 0.5 × GCSat_dB – 1 < PSat_dBm < dBc1out_dBm + 13 - GCSat_dB
     - For *GCType = PSat+GCSat+TOI+dBc1*
       TOIout_dBm_LoLimit =
       dBc1out_dBm + dB_offset – 1 – 0.5 × ((dBc1out_dBm + 13 – GCSat_dB) – PSat_dBm)
       TOIout_dBm_HiLimit =
       dBc1out_dBm + dB_offset + ((dBc1out_dBm + 13 – GCSat_dB) – Psat_dBm) ^{2}
10. Nonlinear model with Rapp Nonlinearity
    - Let $y = c_1 \text{abs}(x) / (1 + (c1*\text{abs}(x) /xs)^{(2*s)})^{(1/(2*s))}$

      where
      - c_1 = *Gain*
      - s > 0; s = nonlinearity smoothness factor = RappS
      - $x_s >= 0$; x_s = input peak voltage at saturation
    - GCSatR = voltage compression ratio = $10^{(GSat / 20)}$

      y_s = output peak voltage at saturation = $PSatv = \sqrt{2\cdot RefR\cdot 10^{\frac{PSat-30}{10}}}$
      x_s = input peak voltage at saturation = PSatv × GCSatR / c_1
    - This nonlinear model was developed for solid state power amplifiers, produces a smooth transition for the AM/AM conversion as the input amplitude approaches

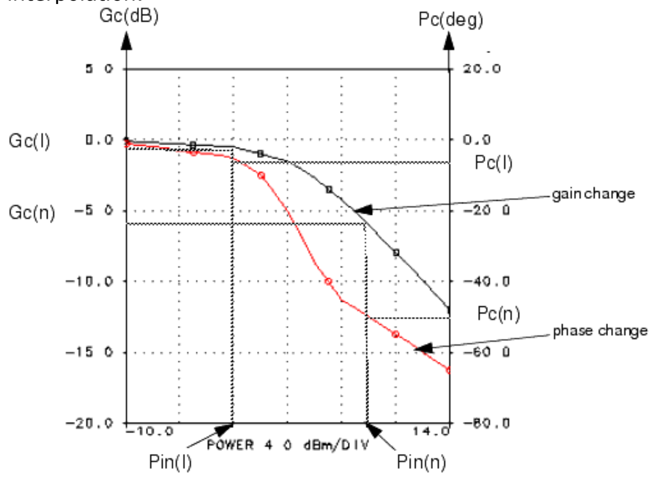
saturation, and has no AM/PM conversion.

- The reference for the Rapp nonlinear model is:
  Rapp C., "Effects of HPA-nonlinearity on a 4-DPSK/OFDM-signal for a digital sound broadcasting system," in Proc. of the Second European Conf. on Satellite Comm.,
  Liège, Belgium, Oct. 1991.

11. Nonlinear model with Gain compression versus input power
    This nonlinear model is defined by the small signal gain, *Gain*, and the array of triple values for input power(dBm), gain change (dB) and phase change (deg) from small signal.
    - Define:
      - dB(G_{ss}) = small signal gain in dB
      - phase(G_{ss}) = small signal phase in deg
      - P = input power in dBm
      - O = output power in dBm
      - OPh = output phase in deg
      - GC = gain change from small signal in dB
        GC[i] = O[i] − P[i] − dB(G_{ss});

        GC[i] = (O[i] − O[i-1]) − (P[i] − P[i-1]) + GC[i-1]
      - PC = phase change from small signal in deg
        PC[i] = OPh[i] − phase(G_{ss});

        PC[i] = (OPh[i] − OPh[i-1]) + PC[i-1];
    - Gain and phase change between the consecutive points is found using linear interpolation.



12. Nonlinear model with AM/AM and AM/PM versus input power
    This nonlinear model is defined by the small signal gain, *Gain*, and the array of triple values for input power(dBm), AM-to-AM (dB/dB), and AM-to-PM (deg/dB).
    The model is achieved by relating to the nonlinear model with Gain compression versus input power. See the prior note for the definition of P, O, OPh, GC and PC
    - Define
      - AM_to_AM = AM = (delta output power in dB)/(delta input power in dB)
      - AM_to_PM = PM = (delta output phase in deg)/(delta input power in dB)
    - By definition
      - AM[i] = 0.5 * (dO[i]/dP[i] + dO[i+1]/dP[i+1])
      - PM[i] = 0.5 * (dOPh[i]/dP[i] + dOPh[i+1]/dP[i+1])
        Where
        - dO[i] = O[i] − O[i-1] with dO[0] = 0;
        - dOPh[i] = OPh[i] − OPh[i-1] with dOPh[0] = 0;
        - dP[i] = P[i] − P[i-1] with dP[0] = dP[1];
    - Given the set of values for AM[i] and PM[i], the associated value for GC[i] and PC[i] are derived. Then the nonlinearity is as defined in the previous note.
13. The noise is defined by the parameters *NoiseFigure* and *RefR*.
    - Let
      - k = Boltzmann constant = 1.3806504e-23 Joules/Kelvin
      - sr = simulation sample rate at this block input
      - factor = 2 for real baseband signal, = 1 for complex envelope signals
      - t0 = reference temperature = 290 Kelvin
      - nf = $10^{(NoiseFigure/10)}$
      - vn_rms = rms noise voltage
    - Then
      - $$vn\_rms = \sqrt{k \cdot (t0 \cdot (nf - 1)) \cdot \frac{sr}{2} \cdot RefR}$$
    - vn_rms is used to generate a Gaussian random variate that is added to the input signal before the signal is amplified through the nonlinearity.

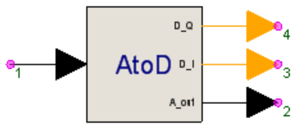
AtoD_ADI Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AtoD_ADI (algorithm)	Analog to Digital Converter for ADI ADC Models

AtoD_ADI



Description: Analog to Digital Converter for ADI ADC Models
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *AtoD ADI Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Model	ADC type A-to-D model: User specified model, AD6645_80.adc, AD6645_105.adc, AD9214_65_2V.adc, AD9214_80_1V.adc, AD9214_105_1V.adc, AD9215_65.adc, AD9215_80.adc, AD9215_105.adc, AD9216_65.adc, AD9216_80.adc, AD9216_105.adc, AD9218_40_1V.adc, AD9218_65_2V.adc, AD9218_80_1V.adc, AD9218_105_1V.adc, AD9219_40.adc, AD9219_65.adc, AD9222_40.adc, AD9222_50.adc, AD9226_2V.adc, AD9228_40.adc, AD9228_65.adc, AD9229_50.adc, AD9229_65.adc, AD9230_170.adc, AD9230_210.adc, AD9230_250.adc, AD9233_105.adc, AD9233_125.adc, AD9236.adc, AD9237_20.adc, AD9237_40.adc, AD9237_65.adc, AD9238_20.adc, AD9238_40.adc, AD9238_65.adc, AD9244_40.adc, AD9244_65.adc, AD9245_20.adc, AD9245_40.adc, AD9245_65.adc, AD9245_80.adc, AD9246_105.adc, AD9246_125.adc, AD9248_20.adc, AD9248_40.adc, AD9248_65.adc, AD9252.adc, AD9254.adc, AD9259.adc, AD9287.adc, AD9289.adc, AD9430_170_LVDS.adc, AD9430_210_LVDS.adc, AD9433_105.adc, AD9433_125.adc, AD9444.adc, AD9445_105_2V.adc, AD9445_105_3p2V.adc, AD9445_125_2V.adc, AD9445_125_3p2V.adc, AD9446_80_2V.adc, AD9446_80_3p2V.adc, AD9446_100_2V.adc, AD9446_100_3p2V.adc, AD9460_80_3p4V.adc, AD9460_105_3p4V.adc, AD9461_125_3p4V.adc, AD9461_130_3p4V.adc, AD9480.adc, AD9600.adc, AD9601.adc, AD9626_170.adc, AD9626_210.adc, AD9626_250.adc, AD9627-11.adc, AD9627.adc, AD9640.adc, AD80141.adc, Ideal_8_Bit.adc, Ideal_10_Bit.adc, Ideal_12_Bit.adc, Ideal_14_Bit.adc	User specified model		Enumeration	NO
ModelDirType	Model directory type: Default, User defined	Default		Enumeration	NO
ModelDir	User defined directory for model adc files			Text	NO
UserModel	User specified A-to-D model			Filename	NO
UserNBits	User specified number of bits in the A-to-D model	8		Integer	NO
UserMinSR	User specified minimum sample rate (encode rate)	0	Hz	Float	NO
UserMaxSR	User specified maximum sample rate (encode rate)	1e12	Hz	Float	NO
UserCommonModeOffset	User specified common mode offset voltage applied to input signal	0	V	Float	NO
UserInputSpan	User specified input signal span	2	V	Float	NO
UserDigitalFormat	User specified output digital format: Offset binary, Twos-complement	Twos-complement		Enumeration	NO
CenterFreq	Spectral center frequency of analog input	0	Hz	Float	NO
NyquistZone	Value representing the analog input Nyquist zone; use n when n*SR/2 < f < (n+1)*SR/2	1		Integer	NO
EnableExtJitter	Enable external jitter: NO, YES	NO		Enumeration	NO
ExtJitter	External RMS jitter applied to the input signal	0	s	Float	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	A_in	input analog signal	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	A_out	output sampled analog signal	envelope	NO
3	D_I	output NBit word as integer for I channel	int	NO
4	D_Q	output NBit word as integer for Q channel	int	NO

Notes/Equations

1. This component models analog-to-digital converters from Analog Devices with reference to Analog Devices part numbers. The input at A_in can be a real baseband signal or a complex envelope signal. The output digital words are in integer form. When the input is a real baseband signal, the output digital word is at the D_I output. When the input is a complex envelope signal, the D_I output contains the digital word for the I envelope and the D_Q ouput contains the digital word for the Q envelope. The A_out contains the quantized form of the input signal. If A_in is a complex envelope signal, then the A_out is also a complex envelope signal at the same characterization frequency.
2. This block reads 1 sample from the input and writes 1 sample to the outputs.
3. ModelDir is the computer disk directory location in which the Analog Devices AtoD model files (*.adc) are located. The default location is in the SystemVue installation directory (typically C:\Program Files\SystemVue(Version)) under Model\ADI. The user can use these model files delivered with SystemVue or use other Analog Devices AtoD model files obtained from Analog Devices and located in another directory.
4. When Model is not 'User specified model'
   The *Model* value is the name of an Analog Devices AtoD model file that is delivered with SystemVue. By selecting one of these predefined model, the full characteristics are defined and used. For these models, the simulator prints out the values for NBits, MinSR, MaxSR, CommonModeOffset, InputSpan and DigitalFormat associated with the selected Analog Devices AtoD model.
5. When Model is 'User specified model'
   The *UserModel* is the Analog Devices *.adc filename that the user separately obtained and would like to use. This option is available when the predefined *.adc filenames are not what the user wants to use and would like to use instead other Analog Devices AtoD models. The *UserNBits*, *UserMinSR*, *UserMaxSR*, *UserCommonModeOffset*, *UserInputSpan*, and *UserDigitalFormat* values are to be entered by the user as is defined for the *UserModel* listed by the user and are value to be obtained from Analog Devices for the model listed.
6. Every simulation sample results in sampling the input A_in and quantization of that input resulting in the output values. When *EnableExtJitter* = YES, then the *ExtJitter* defines the RMS jitter applied to the input A_in before sampling.
7. The Analog Devices AtoD (*.adc) model is defined for use with the input signals in the range [CommonModeOffset, CommonModeOffset + InputSpan]. The CommonModeOffset bias is automatically applied internally to the AtoD_ADI input A_in. Thus, the AtoD_ADI input A_in is defined for use with A_in in the range [-InputSpan / 2, InputSpan / 2), and limited to these limits when A_in is outside this range. The CommonModeOffset and InputSpan are automatically defined when Model is not 'User specified model'. They are defined by *UserCommonModeOffset* and *UserInputSpan* when Model is 'User specified model'.
8. Quantization of the input A_in
   NBits, InputSpan and DigitalFormat are used to quantize the input A_in to generate the digital words that appear at the integer outputs D_I and D_Q and the quantized analog output A_out. These values are either predefined when Model is not 'User specified model', or are user specified when Model is 'User specified model'. Define VRef = InputSpan/2. Define LSB = least significant bit = $2 \times \text{VRef} / 2^{NBits}$.
   The conversion thresholds for A_in are {-VRef + i × LSB}, where i = 1, ... , $2^{NBits} - 1$.
   When DigitalFormat is 'Offset binary', the digital words span integer values in the range [0, 2^ NBits - 1].
   When DigitalFormat is 'Twos-complement', the digital words span integer values in the range [-2^(NBits /2), 2^(NBits /2) - 1].
   A_out output is in the range of [-(VRef - 0.5 LSB), (VRef - 0.5 LSB)] with values {-VRef + (i - 0.5) × LSB}, where i = 1, ... , 2^{NBits}.
9. The Analog Devices AtoD models all have a start up transient time (Latency) during which time the output values are random values.
10. The Analog Devices AtoD models allow the user to specify the spectral center frequency of the analog signal (CenterFreq, when A_in is real) and the analog input Nyquist zone to improve AtoD model behavior.
11. See the Analog Devices AtoD documentation for more detail on their AtoD models. http://www.analog.com/en/analog-to-digital-converters/ad-converters/products/index.html

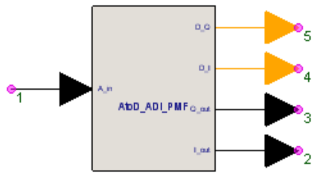
AtoD_ADI_PMF Part

Categories: *Analog/RF* (algorithm), *C++ Code Generation* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
AtoD_ADI_PMF (algorithm)	Analog to Digital Converter for ADI PMF Models

AtoD ADI PMF



Description: Analog to Digital Converter for ADI PMF Models
Domain: Timed
C++ Code Generation Support: YES
Associated Parts: *AtoD ADI PMF Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
PMFModel	PMF type A-to-D model: AD6655.pmf, AD6657.pmf, AD9262.pmf, AD9271.pmf, AD9272.pmf, AD9273.pmf	AD6655.pmf		Enumeration	NO
ModelDirType	Model directory type: Default, User defined	Default		Enumeration	NO
ModelDir	User defined directory for model adc and pmf files			Text	NO
DisplayPMF_Info	Display PMF model information: No, Setup information, Setup information and all available states	No		Enumeration	NO
Set6655_Mode	PMF model 6655 mode: HB Only, HB + FIR, NCO + HB, NCO + HB + FIR, NCO + HB + FIR + fadc/8	HB Only		Enumeration	NO
Set6657_Mode	PMF model 6657 mode: ADC Only, ADC + NS (22%), ADC + NS (33%)	ADC Only		Enumeration	NO
Set9262_Mode	PMF model 9262 mode: 10 MHz Bandwidth, 5 MHz Bandwidth, 2.5 MHz Bandwidth	10 MHz Bandwidth		Enumeration	NO
LowPass	Enable halfband low pass mode: NO, YES	YES		Enumeration	NO
ReverseSpec	Enable spectrum reversal mode: NO, YES	NO		Enumeration	NO
DoubleFIRScale	Double FIR scale factor: NO, YES	NO		Enumeration	NO
DecimationPhase	Decimation phase	0		Integer	NO
NCO_Freq	NCO frequency	0	Hz	Float	NO
NCO_PhaseOffset	NCO phase offset in degrees	0		Float	NO
NoiseShapeTuningWord	Noise shape tuning word	0		Integer	NO
NCO_Kout_m	NCO Kout parameter m	16		Float	NO
LNA_Gain1	LNA scalar gain: Gain1 14, Gain1 15.6, Gain1 18	Gain1 14		Enumeration	NO
LNA_Gain2	LNA scalar gain: Gain2 15.6, Gain2 17.9, Gain2 21.3	Gain2 15.6		Enumeration	NO
VGA_Gain	VGA scalar gain	0		Float	NO
PGA_Gain	PGA scalar gain	0		Float	NO
PGA_Atten	PGA scalar attenuation: Atten 21, Atten 24, Atten 27, Atten 30	Atten 24		Enumeration	NO
LowerCutOffFreq	Lower cutoff frequency: LPF 0, LPF 300000, LPF 700000	LPF 0		Enumeration	NO
AAF_k	AAF k parameter: AAF k 0.7, AAF k 0.8, AAF k 0.9, AAF k 1.0, AAF k 1.1, AAF k 1.2, AAF k 1.3	AAF k 0.7		Enumeration	NO
HPF_CutOffFreq	HPF Cutoff Frequency (not working; to be fixed): HPF Flp/20.7, HPF Flp/11.5, HPF Flp/7.9, HPF Flp/6.0, HPF Flp/4.9, HPF Flp/4.1, HPF Flp/3.5, HPF Flp/3.1, HPF DC	HPF Flp/20.7		Enumeration	NO
LPF_fc_vs_fs	LPF fc percentage of fs: LPF 1/3, LPF 1/4.5	LPF 1/3		Enumeration	NO
CenterFreq	Spectral center frequency of analog input	0	Hz	Float	NO
NyquistZone	Value representing the analog input Nyquist zone; use n when n*SR/2 < f < (n+1)*SR/2	0		Integer	NO
EnableExtJitter	Enable external jitter: NO, YES	NO		Enumeration	NO
ExtJitter	External RMS jitter applied to the input signal	0	s	Float	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	A_in	input analog signal	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	I_out	output sampled analog I signal	envelope	NO
3	Q_out	output sampled analog Q signal	envelope	NO
4	D_I	output NBit word as integer for I channel	int	NO
5	D_Q	output NBit word as integer for Q channel	int	NO

Notes/Equations

1. This models implements a predefined set of Analog Devices parts referenced through their part numbers.
2. At every execution cycle, 1 sample is read from the input and 1 sample is written to the output.
3. ModelDir is the computer disk directory location where the Analog Devices model files (*.pmf) are located. The default location is in the SystemVue installation directory (typically C:\Program Files\SystemVue(Version)) under Model\ADI.
4. For more details on the specific parts refer to their documentation provided by Analog Devices:
    - AD6655: http://www.analog.com/en/rfif-components/digital-updown-converters/ad6655/products/product.html
    - AD6657: http://www.analog.com/en/rfif-components/rxtx-subsystems/ad6657/products/product.html
    - AD9262: http://www.analog.com/en/analog-to-digital-converters/ad-converters/ad9262/products/product.html
    - AD9271: http://www.analog.com/en/analog-to-digital-converters/ad-converters/ad9271/products/product.html
    - AD9272: http://www.analog.com/en/analog-to-digital-converters/ad-converters/ad9272/products/product.html
    - AD9273: http://www.analog.com/en/analog-to-digital-converters/ad-converters/ad9273/products/product.html

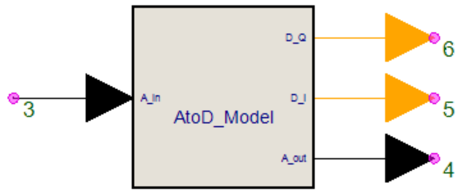
AtoD_Model Part

Categories: *Analog/RF* (algorithm), *C++ Code Generation* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AtoD_Model (algorithm)	Analog to Digital Converter Model

AtoD_Model



Description: Analog to Digital Converter Model
Domain: Timed
C++ Code Generation Support: YES
Associated Parts: *AtoD Model Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
NBits	Number of bits in the A-to-D model	12		Integer	NO
VRef	Reference voltage, -VRef<=input<=VRef	1	V	Float	NO
OutputDigitalFormat	Output digital format: Offset binary, Twos-complement	Twos-complement		Enumeration	NO
DistortionModel	Distortion model: None, Jitter, INL and DNL, ENOB value, SNR and Harmonics, SINAD and SFDR	Jitter, INL and DNL		Enumeration	NO
EnableJitter	Enable jitter: No, Time Domain, Frequency Domain	No		Enumeration	NO
RJrms	Standard deviation of the random jitter	0	s	Float	NO
PhaseNoiseData	Phase noise specification - pairs of offset freq (Hz) and SSB phase noise level (dBc/Hz)	[1000, -30, 10000, -50, 100000, -50]		None	NO
PN_Type	Phase noise model type with random or fixed offset freq spacing and amplitude: Random PN, Fixed Freq Offset, Fixed Freq Offset And Amplitude	Random PN		Enumeration	NO
INL	Integral nonlinearity relative to least significant bit (LSB)	0		Float	NO
DNL	Differential nonlinearity relative to least significant bit (LSB)	0		Float	NO
ENOB	Equivalent number of bits (based on INL and DNL	12		Float	NO
SNR_Model	SNR model: Quantization and Jitter, Quantization and INL/DNL, Quantization and (Jitter or INL/DNL)	Quantization and Jitter		Enumeration	NO
CenterFreq	Spectral center frequency for analog input	0	Hz	Float	NO
Level_dBFS	Signal level in dBFS for analog input	0		Float	NO
SNR_dB	SNR output in dB for analog input	60		Float	NO
H2_dBc	2nd harmonic output level in dBc relative to fundamental output	-400		Float	NO
H3_dBc	3rd harmonic output level in dBc relative to fundamental output	-400		Float	NO
H4_dBc	4th harmonic output level in dBc relative to fundamental output	-400		Float	NO
H5_dBc	5th harmonic output level in dBc relative to fundamental output	-400		Float	NO
SINAD_dB	Output signal to (noise plus harmonic distortion) ratio in dB	60		Float	NO
SFDR_dBc	Output spurious free dynamic range in dBc relative to fundamental output level	70		Float	NO
FFT_Size	FFT size as power of 2: 2^12, 2^13, 2^14, 2^15, 2^16	2^14		Enumeration	NO
ConversionType	Type of input conversion: Clocked, Downsampled	Clocked		Enumeration	NO
Clock	Internal cosine clock frequency	0.2e6	Hz	Float	NO
Phase	Internal clock phase	0.0	deg	Float	NO
DownsampleFactor	Downsampling ratio	1		Integer	NO
DownsamplePhase	Downsampling phase	0		Integer	NO
AntiAliasingFilter	Turn off/on anti-aliasing filter before downsampling: OFF, ON	OFF		Enumeration	NO
ExcessBW	Excess bandwidth of raised cosine anti-aliasing filter	0.5		Float	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	A_in	input analog signal	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	A_out	output sampled analog signal	envelope	NO
3	D_I	output NBit word as integer for I channel	int	NO
4	D_Q	output NBit word as integer for Q channel	int	NO

Notes/Equations

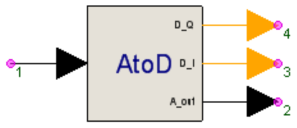
AtoD Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AtoD (algorithm)	Analog to Digital Converter with Integral and Differential Nonlinearities

AtoD



Description: Analog to Digital Converter with Integral and Differential Nonlinearities
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *AtoD Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NBits	Number of bits	8		Integer	NO	[4:16]
VRef	Reference voltage, -VRef<=input<=VRef	1.0	V	Float	NO	(0:∞)
OutputDigitalFormat	Output digital format: Offset binary, Twos-complement	Offset binary		Enumeration	NO	
DistortionModel	Distortion model: None, Jitter/INL/DNL, ENOB value, SNR and Harmonics, SINAD and SFDR	Jitter/INL/DNL		Enumeration	NO	
EnableJitter	Enable jitter: No, Time Domain, Frequency Domain	No		Enumeration	NO	
RJrms	Standard deviation of random jitter	0.0	s	Float	NO	[0:∞)
PhaseNoiseData	Phase noise specification - pairs of offset freq (Hz) and SSB phase noise level (dBc/Hz)			Floating point array	NO	
PN_Type	Phase noise model type with random or fixed offset freq spacing and amplitude: Random PN, Fixed freq offset, Fixed freq offset and amplitude	Random PN		Enumeration	NO	
INL	Integral nonlinearity relative to least significant bit (LSB)	0.0		Float	NO	[DNL/2:∞)
DNL	Differential nonlinearity relative to least significant bit (LSB)	0.0		Float	NO	[0:∞)
ENOB	Equivalent number of bits (based on INL and DNL)	12		Float	NO	[1:16]
SNR_Model	SNR model: Quantization_and_Jitter, Quantization_and_INL_DNL, Quantization_and_Jitter_or_INL_DNL	Quantization_and_Jitter		Enumeration	NO	
CenterFreq	Spectral center frequency for analog input	0.0	Hz	Float	NO	[0.0:∞)
Level_dBFS	Signal level in dBFS for analog input	0.0		Float	NO	(-∞:0.0]
SNR_dB	SNR output in dB for analog input	60.0		Float	NO	(-∞:∞)
H2_dBc	2nd harmonic output level in dBc relative to fundamental output	-400.0		Float	NO	(-ing:-10]
H3_dBc	3rd harmonic output level in dBc relative to fundamental output	-400.0		Float	NO	(-ing:-10]
H4_dBc	4th harmonic output level in dBc relative to fundamental output	-400.0		Float	NO	(-ing:-10]
H5_dBc	5th harmonic output level in dBc relative to fundamental output	-400.0		Float	NO	(-ing:-10]
SINAD_dB	Output signal to (noise plus harmonic distortion) ratio in dB	60.0		Float	NO	(-∞:∞)
SFDR_dBc	Output spurious free dynamic range in dBc relative to fundamental output level	70.0		Float	NO	(-∞:∞)
FFT_Size	FFT size as power of 2: 2^12, 2^13, 2^14, 2^15, 2^16	2^14		Enumeration	NO	
ConversionType	Type of input conversion: Clocked, Downsampled	Clocked		Enumeration	NO	
Clock	Internal cosine clock frequency	0.2e6	Hz	Float	NO	(0:SR]†
Phase	Internal clock phase	0.0	deg	Float	NO	(-∞:∞)
DownsampleFactor	Downsampling ratio	1		Integer	NO	[1:∞)
DownsamplePhase	Downsampling phase	0		Integer	NO	[0:Factor-1]
AntiAliasingFilter	Turn off/on anti-aliasing filter before downsampling: OFF, ON	OFF		Enumeration	NO	
ExcessBW	Excess bandwidth of raised cosine anti-aliasing filter	0.5		Float	NO	[0:1]

Input Ports

Port	Name	Description	Signal Type	Optional
1	A_in	input analog signal	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	A_out	output sampled analog baseband signal	envelope	NO
3	D_I	output NBit word as integer for I channel	int	NO
4	D_Q	output NBit word as integer for Q channel	int	NO

Notes/Equations

1. This model implements an analog-to-digital converter (ADC) with an internal clock. It can implement an ideal ADC or integral and differential non-linearities as well as jitter or phase noise for the internal clock can be specified by the user to model a more realistic ADC. Offset and gain errors are not included in this model.
2. The input *A_in* can be a real baseband signal or a complex envelope signal.

- When the input signal is a real baseband signal, it is sampled (with the appropriate jitter/phase noise applied) and quantized (based on the *NBits, VRef, INL, and DNL*) to generate the real baseband output signal *A_out*. The bits that represent the quantized input signal value (the index of the quantization level to which the input signal sample was quantized to) are output as an integer value at the port *D_I*. The signal at the port *D_Q* is always 0.
- When the input signal is a complex envelope signal, its *I* and *Q* envelopes are sampled (with the appropriate jitter/phase noise applied) and quantized (based on the *NBits, VRef, INL, and DNL*) to generate the *I* and *Q* envelopes of the output signal *A_out*. The characterization frequency of *A_out* is the same as the characterization frequency of *A_in*. The bits that represent the quantized values of the *I* and *Q* envelope of the input signal (the indices of the quantization levels to which the envelope values were quantized to) are output as integer values at the ports *D_I* and *D_Q* respectively.

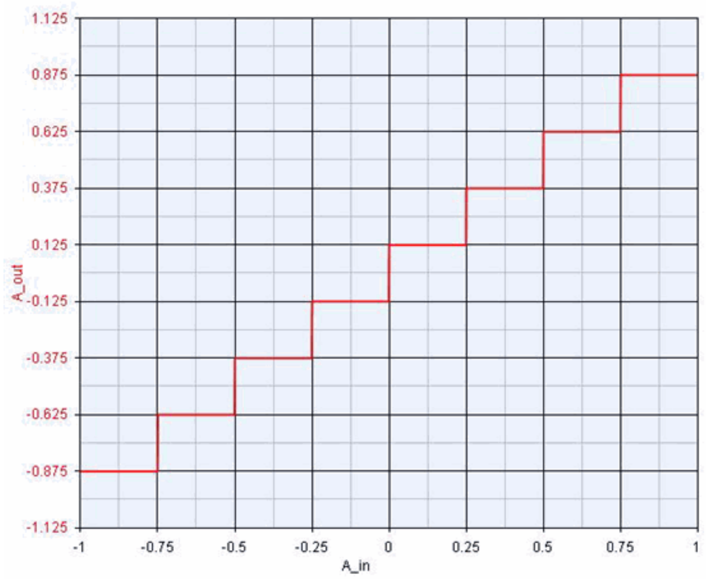
3. The *NBits* parameter sets the number of bits for the ADC. The number of quantization levels is 2^{NBits}. The higher the number of bits the smaller the quantization error, which means that the output signal *A_out* matches more closely with the input *A_in*.
4. The *VRef* parameter sets the reference voltage for the ADC. The input signal must lie in the range [−*Verf, VRef*]. Otherwise, clipping occurs.
5. The *INL* and *DNL* parameters set the integral and differential nonlinearities relative to the LSB (Least Significant Bit) for the ADC.
6. The DNL error is defined as the maximum difference between the ideal ADC step width of 1 LSB (1 LSB = $2 \cdot VRef/2^{NBits}$) and the actual ADC step width (after the offset and gain errors have been compensated for). A DNL error specification of less than 1 LSB guarantees that there will be no missing codes in the ADC transfer function.

   Note
   - When DNL > 0, there is no guarantee that this model will actually use a transfer function with the requested DNL error.
   - The DNL error is modeled by a Gaussian distribution. There is approximately 1% probability that the random Gaussianly distributed DNL errors will be equal to or greater than the *DNL* parameter value (or less than −*DNL*).
   - If that happens the actual error is truncated to *DNL* (or −*DNL*) and the ADC transfer function will have the DNL error set by the user.
   - Otherwise, the ADC transfer function will have a DNL error smaller than what the user has set.
   - The higher the *NBits* value the more steps the ADC has and therefore the more likely the DNL error for one of those steps will be equal to or greater than the *DNL* parameter value (or less than −*DNL*), which will result in an ADC transfer function with the exact desired DNL.

7. The INL error is defined as the maximum deviation (in LSB) between the ideal ADC transfer function (straight line) and the actual ADC transfer function (after the offset and gain errors have been compensated for).

   Note
   - When INL > 0, there is no guarantee that this model will actually use a transfer function with the requested INL error.
   - The INL error is the cumulative sum of the DNL errors, which (as explained earlier) are modeled by Gaussian distribution.
   - Therefore, the probability the probability that the actual ADC transfer function will have an INL error equal to the *INL* parameter value (or −*INL*) depends on the values of the *NBits*, *INL*, and *DNL* parameters.
   - With larger *NBits*, smaller *INL*, and larger *DNL*, the more likely the the actual ADC transfer function will have an INL error equal to the *INL* parameter value (or −*INL*).

8. For an ideal ADC (INL=0 and DNL=0) the input quantization thresholds are $-VRef + i \cdot LSB$ ($i = 1, \ldots, 2^{NBits} - 1$) and the output quantization levels are $-VRef + (i - 0.5) \cdot LSB$ ($i = 1, \ldots, 2^{NBits}$), where $LSB = 2 \cdot VRef/2^{NBits}$. The following graph shows the transfer function of an ideal 3-bit ADC with *VRef* = 1. Based on the equations given above, the quantization thresholds (x-axis transition points) are $-1 + i \cdot 0.25$ ($i = 1, \ldots, 7$) = { −0.75, −0.5, −0.25, 0.0, 0.25, 0.5, 0.75 } and the quantization levels (y-axis transition points) are $-1 + (i - 0.5) \cdot 0.25$ ($i = 1, \ldots, 8$) = { −0.875, −0.625, −0.375, −0.125, 0.125, 0.375, 0.625, 0.875 }.



9. This ADC model can operate in two different modes: *Clocked* and *Downsampled* (specified in the *ConversionType* parameter)
   - In the ***Clocked* mode**, the model reads one sample from its input and writes one sample to each of its outputs. The internal clock frequency (*Clock* parameter) and phase (*Phase* parameter) need to be set. The internal clock is of the form cos(2·π·*Clock·t* + *Phase*). The input signal is sampled at the positive zero crossings of the clock. To be able to detect zero crossings, at least four samples per clock period are needed. Therefore, when *SR*/4 < *Clock* ≤ *SR*, the internal clock is represented at 4·*SR* (*SR* is the sampling rate of the input signal). This does not affect the output signal sampling rate, which is always the same as the input signal sampling rate (*SR*). Once the input signal is sampled, the sampled value is quantized and held constant at the output until the next clock positive zero crossing occurs. If *Clock* < *SR*/2, the output will have the form of a piece wise constant waveform because of the sample and hold operation. A piece wise constant waveform can show at the output even when *Clock* = *SR* if *NBits* is not big enough, which will result in multiple input signal values to be quantized to the same output quantization level.
   - In the ***Downsampled* mode**, the *DownsampleFactor* and *DownsamplePhase* parameters need to be set. In this case, the model reads *DownsampleFactor* samples from its input and writes 1 sample to each of its outputs. The sample written to the outputs is the quantized *DownsamplePhase*-th sample in the block of *DownsampleFactor* input samples. No explicit clock signal is used but the equivalent clock signal is cos(2·π·*SR·t/DownsampleFactor* + 2·π·*DownsamplePhase/DownsampleFactor* − π/2). Since the input signal is downsampled, the output signal will not have the form of a piece wise constant waveform (no sample and hold operation), unless *NBits* is not big enough, which will result in multiple input signal values to be quantized to the same output quantization level.
     When downsampling a signal, aliasing can occur (if the sampling rate of the downsampled signal is not high enough to represent the frequency content of the original signal). Therefore, a raised cosine antialiasing filter can be set up (set *AntiAliasingFilter* parameter to *ON*). The *ExcessBW* (or roll off factor) of this filter can also be specified. If the antialiasing filter is used a delay is introduced in the output signals.

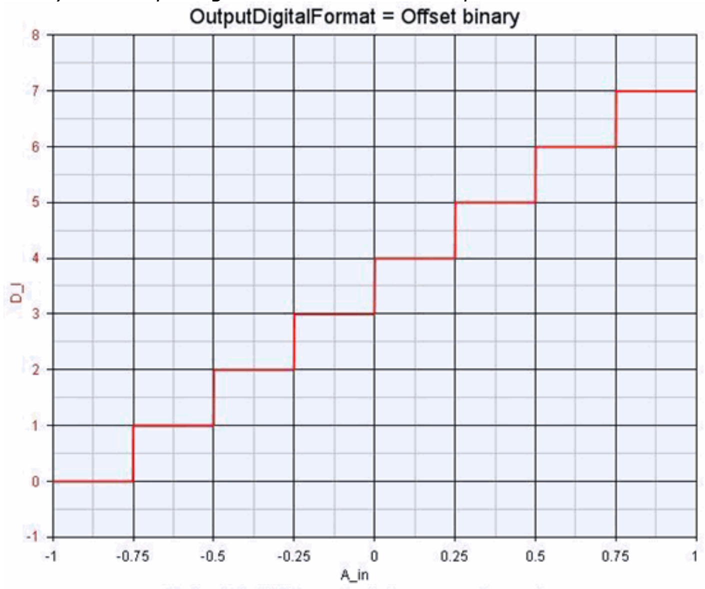
     For more details on downsampling see the documentation of the *DownSampleEnv* (algorithm) model.

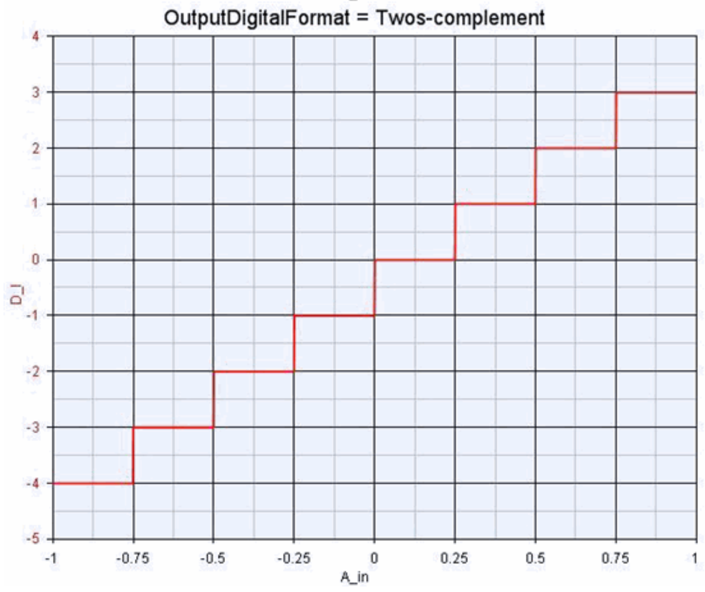
10. The internal clock whether explicit (*Clocked* mode) or implicit (*Downsampled* mode) can be impaired with jitter (set *EnableJitter* parameter to *Time Domain*) or phase noise (set *EnableJitter* parameter to *Frequency Domain*). Jitter or phase noise affect the clock zero crossings and therefore the instant the input signal gets sampled.
    - When *EnableJitter* = *Time Domain*, the standard deviation of the random jitter can be specified in the *RJrms* parameter. Jitter is modeled using a Gaussian distribution with zero mean and *RJrms* standard deviation.
    - When *ConversionType* = *Clocked*, then 3·*RJrms* (99.7% of a Gaussian distribution lies within 3·σ of its mean) has to be smaller than 1/4 of the clock time step (note that when *SR*/4 < *Clock* ≤ *SR*, the clock time step is 1/4 of the input signal time step).
    - When *ConversionType* = *Downsampled*, then 3·*RJrms* has to be smaller than 1/2 of the input signal time step.
      If the generated random jitter value happens to be outside the range [−3·*RJrms* , 3·*RJrms*] it gets limited to ±3·*RJrms*.
    - When *EnableJitter* = *Frequency Domain*, the phase noise specification can be entered in the *PhaseNoiseData* parameter as pairs of offset freq in Hz and SSB

(Single Side Band) phase noise power level in dBc/Hz (with respect to a 1 Volt amplitude cosine wave on 50 Ohms). The *PN_Type* parameter provides additional options on how phase noise is modeled. For more details about phase noise modeling see the documentation of the *Oscillator* (algorithm) model.

- When *ConversionType* = *Clocked*, the phase noise error introduced cannot be greater (in absolute value) than the clock phase increment corresponding to 1/4 of the clock time step (note that when *SR*/4 < *Clock* ≤ *SR*, the clock time step is 1/4 of the input signal time step).
- When *ConversionType* = *Downsampled*, the phase noise error introduced cannot be greater (in absolute value) than the clock phase increment corresponding to 1/2 of the input signal time step.
  If the generated phase noise error value happens to be outside the ranges mentioned above it gets limited to the upper or lower limits of the acceptable ranges.

11. The *OutputDigitalFormat* parameter sets the range of values for the *D_I* and *D_Q* outputs. If *OutputDigitalFormat* = *Offset binary*, then *D_I* and *D_Q* are integers in the range [0, $2^{NBits}-1$]. If *OutputDigitalFormat* = *Twos-complement*, then *D_I* and *D_Q* are integers in the range [$-2^{NBits-1}$, $2^{NBits-1}-1$]. The following graphs show the *D_I* output for an ideal 3-bit ADC with *VRef* = 1, when *OutputDigitalFormat* = *Offset binary* and *OutputDigitalFormat* = *Twos-complement*.





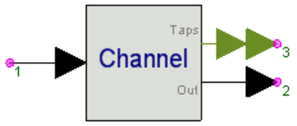
CommsChannel Part

Categories: *Analog/RF* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
CommsChannel (algorithm)	Wireless Channel Model

CommsChannel



Description: Wireless Channel Model
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *CommsChannel Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
ModelType	Channel model type: UserDefined, Pedestrian_A, Pedestrian_B, Vehicular_A, Vehicular_B, Extended_Pedestrian_A, Extended_Vehicular_A, Extended_TypicalUrban	UserDefined		Enumeration	NO	
Delay	User defined tap delays in usec	[0.0 , 0.03, 0.15, 0.31 ,0.37, 0.71 , 1.09 , 1.73 , 2.51]		Floating point array	NO	[0:10000.0]
Power	User defined relative tap powers in dB	[0.0, -1.5, -1.4, -3.6, -0.6, -9.1, -7.0, -12.0, -16.9]		Floating point array	NO	(-∞:0]
RiceanFactor	User defined tap Ricean K-factors in linear scale	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]		Floating point array	NO	[0.0:1000.0]
Velocity	Velocity of mobile station in km/hour	120		Float	NO	[0.001:1000]
PathLoss	Include large-scale pathloss: NO, YES	NO		Enumeration	NO	
PropDistance	Distance between the transmit and receive stations	1000	m	Float	NO	[200:5000]
PwrNormal	Normalize output power: NO, YES	NO		Enumeration	NO	
PwrMeasPeriod	Power measurement time period for output power normalization	0.001	s	Float	NO	(0:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	channel input signal	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	channel output signal	envelope	NO
3	tap	fading channel complex tap coefficients	multiple complex	NO

Notes/Equations

1. This model is used to generate channel models for mobile wireless applications.
2. This model reads 1 sample at its input and writes 1 sample at all outputs.
3. Model use of *Velocity*, *PropDistance*, *PathLoss*, *PwrNormal* and *PwrMeasPeriod*
   Velocity specifies the mobile's velocity relative to base station.
   PropDistance specifies the distance between base station and mobile station.
   PathLoss identifies whether the large-scale pathloss is included.
   If *PathLoss* = *NO*, then the path loss is not included in this model and the parameters describing the environment are unused.

If *PathLoss* = *YES*, then the path loss for both urban and suburban environments is modeled by the COST 207 model with a correction term. There are three terms which make up the model:

The Path Loss model for outdoor to indoor and pedestrian test environment is,

$$L = 40\log R + 30\log f + 49$$

where R is the propagation distance and f is the frequency.

The Path Loss model for vehicular test environment is,

$$L = [40(1 - 4 \times 10^{-3}\Delta h_b)]\log R - 18\log\Delta h_b + 21\log f + 80$$

where R is the propagation distance and f is the frequency, Δh_b is the height between base station antenna and mobile.

PwrNormal specifies whether the ouput power is normalized to the input power. When *PwrNormal* = YES, then the output power is normalized using the *PwrMeasPeriod* which is the time period for measuring power to ensure that the output power is equal to the input power with the resolution of PwrMeasPeriod. The method for adjusting the ouput power is that, assuming the power level measured in two contiguous time periods with *PwrMeasPeriod* seconds is identical, a gain, calculated on the power level measured in the (i-1)th time period, is multiplied to the signal in the ith time period to ensure that the output power in the ith time period is equal to the input power, and so on for the (i+1)th time period. Hence the first time period with *PwrMeasPeriod* seconds remain unadjusted.

4. Channel model definition when *ModelType* = Extended_Pedestrian_A, Extended_Vehicular_A, Extended_TypicalUrban

This model is implemented following the channel model requirements defined for 3GPP LTE mobile wireless applications and based onR4-070872 3GPP TR 36.803v0.3.0. See references 1 and 2 below.

For this model, the *Delay*, *Power* and *RiceanFactor* are predefined and not user specifiable.

For this model a delay of 64 tokens is introduced in the outputs.

A set of 3 channel models are implemented to simulate the multipath fading propagation conditions. The multipath fading is modeled as a tapped-delay line with a number of taps at fixed positions on a sampling grid. The gain associated with each tap is characterized by a distribution (Ricean with a K-factor>0, or Rayleigh with K-factor=0) and the maximum Doppler frequency that is determined from the mobile speed. For each tap, the method of filtered noise is used to generate channel coefficients with the specified distribution and spectral power density.

The definition of the 3 specific channels are shown in the following tables:

Extended Pedestrian A model (EPA)

Tap	Excess tap delay [ns]	Relative power [dB]
1	0	0.0
2	30	-1.0
3	70	-2.0
4	90	-3.0
5	110	-8.0
6	190	-17.2
7	410	-20.8

Extended Vehicular A model (EVA)

Tap	Excess tap delay [ns]	Relative power [dB]
1	0	0.0
2	30	-1.5
3	150	-1.4
4	310	-3.6
5	370	-0.6
6	710	-9.1
7	1090	-7.0
8	1730	-12.0
9	2510	-16.9

Extended Typical Urban model (ETU)

Tap	Excess tap delay [ns]	Relative power [dB]
1	0	-1.0
2	50	-1.0
3	120	-1.0
4	200	-0.0
5	230	-0.0
6	500	-0.0
7	1600	-3.0
8	2300	-5.0
9	5000	-7.0

The total channel gain is normalized by adding the specified Normalization Factor to each tap.

The Doppler spectrum is modelled using the well known Clarke or Classical Doppler spectrum. The power spectral density (PSD) function is defined as follows: !3gpplte-channel-2008-1-01.gif! where !3gpplte-channel-2008-1-02.gif! is the net power, and !3gpplte-channel-2008-1-03.gif! denotes the maximum Doppler frequency, and !3gpplte-channel-2008-1-04.gif! where !3gpplte-channel-2008-1-06.gif! is the speed of the mobile, !3gpplte-channel-2008-1-05.gif! is the carrier frequency and ***c*** is the speed of light.

5. Channel model definition when *ModelType* = Pedestrian_A, Pedestrian_B, Vehicular_A, Vehicular_B
   This model is implemented following Rec.ITU-R M.1225 for mobile wireless applications. See references 3 below.
   For this model, the *Delay*, *Power* and *RiceanFactor* are predefined and not user specifiable.
   For this model a delay of 64 tokens is introduced in the outputs.
   A set of 4 modified International Telecommunication Union (ITU) channel models are constructed to simulate the multipath fading of the channel. The multipath fading is modeled as a tapped-delay line with 6 taps with non-uniform delays. The gain associated with each tap is characterized by a distribution (Ricean with a K-factor>0, or Rayleigh with K-factor=0) and the maximum Doppler frequency. For each tap, the method of filtered noise is used to generate channel coefficients with the specified distribution and spectral power density.
   The definition of the 4 specific ITU channels is shown in the following tables:

Outdoor to Indoor and Pedestrian Test Environment Tapped-Delay-Line Parameters

Tap	Channel A		Channel B		Doppler Spectrum
	Relative Delay (ns)	Average Power (dB)	Relative Delay (ns)	Average Power (dB)	
1	0	0	0	0	Classic
2	110	-9.7	200	-0.9	Classic
3	190	-19.2	800	-4.9	Classic
4	410	-22.8	1200	-8.0	Classic
5	ºTMºTM	ºTMºTM	2300	-7.8	Classic
6	ºTMºTM	ºTMºTM	3700	-23.9	Classic

Vehicular Test Environment Tapped-Delay-Line Parameters

Tap	Channel A		Channel B		Doppler Spectrum
	Relative Delay (ns)	Average Power (dB)	Relative Delay (ns)	Average Power (dB)	
1	0	0.0	0	-2.5	Classic
2	310	-1.0	300	0	Classic
3	710	-9.0	8900	-12.8	Classic
4	1090	-10.0	12900	-10.0	Classic
5	1730	-15.0	17100	-25.2	Classic
6	2510	-20.0	20000	-16.0	Classic

   The total channel gain is normalized by adding the specified Normalization Factor to each tap.
   The specified Doppler is the maximum Doppler frequency parameter (fm) of the rounded spectrum which has the power spectral density (PSD) function as follows:

$$S(f) = \begin{cases} \dfrac{1}{\pi\sqrt{1-f_0^2}} & |f_0| \le 1 \\ 0 & |f_0| > 1 \end{cases}$$

   where $f_0 = \dfrac{f}{f_m}$ and $f_m = \dfrac{v}{c}f$, v is the mobile's velocity relative to base station.
   The set of ITU channel models specify statistical parameters of microscopic effects. To simulate the real channel, these statistics have to be combined with macroscopic channel effects, i.e. the path loss (including shadowing) which will be introduced in the later section.
   The COST 207 model with a correction term is used to simulate the path loss for both pedestrian and vehicular environments if the PathLoss is ON and other parameters are set according to the specific environment.
6. Channel model definition when *ModelType* = UserDefined
   For this model, the *Delay*, *Power* and *RiceanFactor* array values specified are used to define the model. These three arrays must has the same size. This size defines N taps in a tapped-delay line. The multipath fading model follows the same approach for the above predefined models, but with user defined characteristics. The multipath fading is modeled as a tapped-delay line with N taps with the delays (*Delay*) and gains (*Power*) specified. Each gain will have a distribution defined by the the *RiceanFactor* (Ricean when *RiceanFactor* > 0, or Rayleigh when *RiceanFactor* = 0) and the maximum Doppler frequency. For each tap, we use the method of filtered noise to generate channel coefficients with the specified distribution and spectral power density.

References

1. Rec.ITU-R M.1225, *Guidelines For Evaluation Of Radio Transmission Technologies For IMT-2000, 1997*.
2. 3GPP TS 36.104 v8.1.0, User Equipment (UE) radio transmission and reception, 2008-03.
3. Rec.ITU-R M.1225, *Guidelines For Evaluation Of Radio Transmission Technologies For IMT-2000, 1997* .

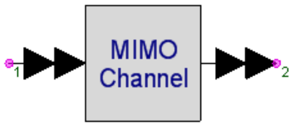
CommsMIMO_Channel Part

Categories: *Analog/RF* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
LTE_MIMO_Channel (algorithm)	LTE MIMO Channel Model

LTE_MIMO_Channel



Description: LTE MIMO Channel Model
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *CommsMIMO Channel Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
AntennaConfig	Antenna configuration as (number of Tx) x (number of Rx) antennas: TR_1x2, TR_2x2, TR_4x2, TR_4x4	TR_2x2		Enumeration	NO	
CorrelationType	MIMO channel correlation matrix type: Low, Medium, High	Low		Enumeration	NO	
ModelType	Channel model type: UserDefined, Extended_Pedestrian_A, Extended_Vehicular_A, Extended_TypicalUrban	UserDefined		Enumeration	NO	
Delay	User defined tap delays in usec	[0.0 , 0.03, 0.15, 0.31 ,0.37, 0.71 , 1.09 , 1.73 , 2.51]		Floating point array	NO	[0:10000.0]
Power	User defined relative tap powers in dB	[0.0, -1.5, -1.4, -3.6, -0.6, -9.1, -7.0, -12.0, -16.9]		Floating point array	NO	(-∞:0]
RiceanFactor	User defined tap Ricean K-factors in linear scale	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]		Floating point array	NO	[0.0:1000.0]
Velocity	Velocity of mobile station in km/hour	120		Float	NO	[0.001:1000]

Input Ports

Port	Name	Description	Signal Type	Optional
1	TxSig	Signals supplied to transmit array	multiple envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	RxSig	Signals at output of receive array	multiple envelope	NO

Notes/Equations

1. The model implements a MIMO channel as defined for 3GPP LTE wireless systems and follows the definition in B.2 of 36.101 [1].
2. This model reads 1 sample from the inputs and writes 1 sample to the outputs.
3. The multipath propagation conditions consist of several parts:
   - delay profile in the form of a "tapped delay-line", characterized by a number of taps at fixed positions on a sampling grid. The profile can be further characterized by the r.m.s. delay spread and the maximum delay spanned by the taps.
   - A combination of channel model parameters that include the Delay profile and the Doppler spectrum, that is characterized by a classical spectrum shape and a maximum Doppler frequency
   - A set of correlation matrices defining the correlation between the UE and eNodeB antennas in case of multi-antenna systems.

4. ModelType parameter specifies the propagation model selected from Extended Pedestrian A, Extended Vehicular A, Extended Typical Urban and user-defined model. Each pre-defined propagation model defines the number of channel taps, delay spread and Relative power for each tap, as shown in Table B.2.1-1, Table B.2.1-2, Table B.2.1-3 and Table B.2.1-4 of 36.101 [1]. For the user-defined model, the power delay profiles (PDP) is defined by the user as follows:
   - *Delay*, *Power* and *RiceanFactor* parameters specify the delay, power and ricean factor for each path when ModelType is selected as UserDefined.
5. AntennasConfig defines the antenna configurations at eNodeB and UE respectively. The antenna configurations are in the format of MxN, where M is the number of antennas at eNodeB, and N is the number of antennas at UE. The bus width at the input port should be equal to M, while the bus width at the input port should be equal to M should be equal to M. Currently the allowable configurations are 1x2, 2x2, 4x2, and 4x4.
6. CorrelationType defines the spatial correlation between the antennas at the eNodeB and UE, selected from Low, Medium, and High.
   For each antenna configuration, the channel correlation matrix is shown as follows,

1x2 case	$R_{spat} = R_{UE} = \begin{bmatrix} 1 & \beta \\ \beta^* & 1 \end{bmatrix}$
2x2 case	$R_{spat} = R_{eNB} \otimes R_{UE} = \begin{bmatrix} 1 & \alpha \\ \alpha^* & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & \beta \\ \beta^* & 1 \end{bmatrix} = \begin{bmatrix} 1 & \beta & \alpha & \alpha\beta \\ \beta^* & 1 & \alpha\beta^* & \alpha \\ \alpha^* & \alpha^*\beta & 1 & \beta \\ \alpha^*\beta^* & \alpha^* & \beta^* & 1 \end{bmatrix}$
4x2 case	$R_{spat} = R_{eNB} \otimes R_{UE} = \begin{bmatrix} 1 & \alpha^{1/9} & \alpha^{4/9} & \alpha \\ \alpha^{1/9*} & 1 & \alpha^{1/9} & \alpha^{4/9} \\ \alpha^{4/9*} & \alpha^{1/9*} & 1 & \alpha^{1/9} \\ \alpha^* & \alpha^{4/9*} & \alpha^{1/9*} & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & \beta \\ \beta^* & 1 \end{bmatrix}$
4x4 case	$R_{spat} = R_{eNB} \otimes R_{UE} = \begin{bmatrix} 1 & \alpha^{1/9} & \alpha^{4/9} & \alpha \\ \alpha^{1/9*} & 1 & \alpha^{1/9} & \alpha^{4/9} \\ \alpha^{4/9*} & \alpha^{1/9*} & 1 & \alpha^{1/9} \\ \alpha^* & \alpha^{4/9*} & \alpha^{1/9*} & 1 \end{bmatrix} \otimes \begin{pmatrix} 1 & \beta^{1/9} & \beta^{4/9} & \beta \\ \beta^{1/9*} & 1 & \beta^{1/9} & \beta^{4/9} \\ \beta^{4/9*} & \beta^{1/9*} & 1 & \beta^{1/9} \\ \beta^* & \beta^{4/9*} & \beta^{1/9*} & 1 \end{pmatrix}$

   where, ɑ and β are channel correlation factor at the eNodeB and UE respectively. ɑ and β for different correlation types are given in the following table according to CorrelationType parameter.

Low correlation	Medium Correlation	High Correlation
ɑ=0, β=0	ɑ=0.3, β=0.9	ɑ=0.9, β=0.9

7. In this model, the channel correlation matrices are per-tap applied on each tap independently.
8. Velocity specifies the mobile unit's velocity (v) relative to the base station, in units of kilometer/hour. The specified Doppler is the maximum Doppler frequency parameter (fm) of the rounded spectrum which has the classical spectrum shape as follows:
   $$S(f) = \begin{cases} \dfrac{1}{\pi\sqrt{1-f_0^2}} & \lvert f_0 \rvert \le 1 \\ 0 & \lvert f_0 \rvert > 1 \end{cases}$$
   where $f_0 = \dfrac{f}{f_m}$ and $f_m = \dfrac{v}{c}f$.
9. Output delay: a delay of 64 tokens is introduced in this model

References

1. 3GPP TS 36.101 v8.5.1 "User Equipment (UE) radio transmission and reception", March 2009.

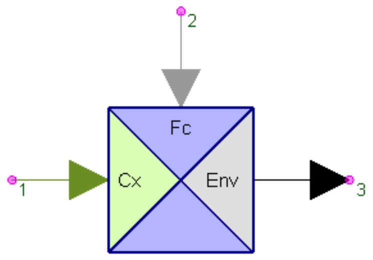
CxToEnv Part

Categories: *Analog/RF* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
CxToEnv (algorithm)	Complex to Envelope

CxToEnv



Description: Complex to Envelope
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *CxToEnv Part* (algorithm)

Model Parameters

Name	**Description**	**Default**	**Units**	**Type**	**Runtime Tunable**	**Range**
Fc	Characterization frequency	0.2e6	Hz	Float	NO	(0:∞)

Input Ports

Port	**Name**	**Signal Type**	**Optional**
1	input	complex	NO
2	fc	envelope	YES

Output Ports

Port	**Name**	**Signal Type**	**Optional**
3	output	envelope	NO

Notes/Equations

1. The CxToEnv block converts the complex signal at *input* to a complex envelope signal at *output* using the characterization frequency associated with the complex envelope signal at input *fc*.
2. This block reads 1 sample from both inputs and writes 1 sample to *output*.
3. The input *fc* is optional. When not connected, the value for the parameter *Fc* is used as the value for *fc* (f2) in the following discussion.
4. Define the *input* signal at pin 1 as $v_1 = v_{i_1}(t) + jv_{q_1}(t)$
5. The signals at *fc* (pin k = 2) and at *output* (pin k = 3) have complex envelope representation.
   - $v_k = v_k(t)\exp(j2\pi f_k t)$, where $v_k(t) = v_{i_k}(t) + jv_{q_k}(t)$
6. The *output* is
   - $v_3 = v_{i_1}(t) + jv_{q_1}(t)$ with $f_3 = f_2$
7. CxToEnv is a modulator whose *output* obtains its I and Q values from the *input* and its carrier frequency from *fc*.
8. If the *fc* input is not a complex envelope signal, then the *output* will be made a real signal and the imaginary part of *input* will be ignored.

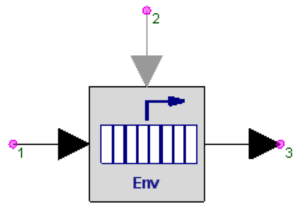
DelayEnv Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
DelayEnv (algorithm)	Envelope Delay

DelayEnv



Description: Envelope Delay
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *DelayEnv Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Delay	Time delay (used with no control input is connected)	0.0	s	Float	NO	[0*TStep:∞)†
InterpolationMethod	Signal interpolation method: None, Linear, Lagrange	None		Enumeration	NO	
IncludeCarrierPhaseShift	Include RF carrier phase shift: NO, YES	YES		Enumeration	NO	
MaxTimeDelay	Maximum time delay available (used when the control input is connected)	0.0	s	Float	NO	[0*TStep:∞)†

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	envelope	NO
2	control	control signal	envelope	YES

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output signal	envelope	NO

Notes/Equations

1. DelayEnv delays the input signal by the amount *Delay*.
2. This model reads 1 sample from the inputs and writes 1 sample to the output.
3. The *IncludeCarrierPhaseShift* value is not used when the input is a real signal. It is only used when the input is a complex envelope signal.
4. When *InterpolationMethod* is *Linear* or *Lagrange*, the input is interpolated to achieve the specified time delay.
   - When *InterpolationMethod* is *Lagrange* and delay is less than 2 /T_S , then *Linear* interpolation is used where T_S is the sampling interval of the input signal
5. The input *control* is optional. When connected, the value for the parameter *Delay* is ignored and the value at the *control* pin is used instead.
   - In the following discussion, 'delay' implies either the *Delay* value (when there is no *control* input), or the *control* value (when there is a *control* input).
   - When the *control* input is used, then the delay is limited to the MaxTimeDelay value.
6. Let
   - T′ = T_S round (delay /T_S) where T_S is the sampling interval of the input signal, when InterpolationMethod is *None*
   - T′ = delay, when InterpolationMethod is *Linear* or *Lagrange*
7. Representation of the delayed *output* signal when the input is a real signal.
   - The input signal v1(t) is a real value.
   - Then
     - $v_2(t) = v_1(t - T')$
8. Representation of the delayed *output* signal when the input is a complex envelope

signal.

- The input signal V $_1$ (t) is represented by its in-phase and quadrature components about its carrier frequency, f $_c$.

  $V_1(t) = \Re\left(v_1(t)\exp(j2\pi f_{c_1}t)\right)$

  where

  $v_1(t) = v_{I_1}(t) + jv_{Q_1}(t)$

- Then

  . $V_2(t) = \Re\left(v_2(t)\exp(j2\pi f_{c_2}t)\right)$

  where

  $f_{c_2} = f_{c_1}$

  if *IncludeCarrierPhaseShift* = NO, then

  .

  $v_2(t) = v_1(t - T')$

  if *IncludeCarrierPhaseShift* = YES, then

  .

  $v_2(t) = v_1(t - T')\exp(-j2\pi f_{c_1}T')$

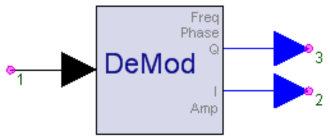
Demodulator Part

Categories: *Analog/RF* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Demodulator (algorithm)	Demodulator

Demodulator (Demodulator)



Description: Demodulator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Demodulator Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
OutputType	Output type: I/Q, Amp/Phase, Amp/Freq	I/Q		Enumeration	NO		
FCarrier	Carrier frequency	0.2e6	Hz	Float	NO	(0:∞)	f_c
InitialPhase	Initial phase	0	deg	Float	NO	(-∞:∞)	θ
AmpSensitivity	Amplitude sensitivity	1		Float	NO	(-∞:∞)	S_a
PhaseSensitivity	Phase deviation sensitivity in Volts/degree	1.0/90.0		Float	NO	(-∞:∞)	S_p
FreqSensitivity	Frequency deviation sensitivity in Volts/Hz	1.0e-4		Float	NO	(-∞:∞)	S_f
MirrorSignal	Mirror signal about carrier: NO, YES	NO		Enumeration	NO		
ShowIQ_Impairments	Show I and Q impairments: NO, YES	NO		Enumeration	NO		
GainImbalance	Gain imbalance in dB, Q channel relative to I channel	0.0		Float	NO	(-∞:∞)	G
PhaseImbalance	Phase imbalance, Q channel relative to I channel	0.0	deg	Float	NO	(-∞:∞)	φ
I_OriginOffset	I origin offset	0.0		Float	NO	(-∞:∞)	I_{off}
Q_OriginOffset	Q origin offset	0.0		Float	NO	(-∞:∞)	Q_{off}
IQ_Rotation	IQ rotation	0.0	deg	Float	NO	(-∞:∞)	R

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	complex envelope vector input	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output1	output1	real	NO
3	output2	output2	real	NO

Notes/Equations

1. The Demodulator model implements a coherent demodulator that can be used to perform amplitude, phase, frequency, or I/Q demodulation.
2. This model reads 1 sample from the input and writes 1 sample to the outputs.
3. The input signal must be a complex envelope signal with characterization frequency greater than zero. Real input signals are not allowed by this model. If use with a real signal is needed, then the input to this model can be preceded with an *EnvFcChange* (algorithm) model that will recharacterize a real signal to its representation at a specified frequency.
4. Let's assume that the complex envelope input signal is cx = I + j·Q at a characterization frequency of *f* , which represents the real signal I·cos(2π*f* t) −

1 1

Q·sin(2πf_1t).

5. First the *InitialPhase* is applied to the input signal by rotating it by −θ (cx = cx·(cos(θ) − j·sin(θ)).
6. Then cx is recharacterized at f_c.
    - See the documentation for *EnvFcChange* (algorithm) for detail on how a signal is converted from one characterization frequency to another.
    - The following discussion is in context with the input signal after it has been characterized at f_c, that is, cx now represents the new complex envelope at f_c.
7. Then the IQ impairments are applied to the input.
    - *I_OriginOffset* and *Q_OriginOffset*: cx = cx + (I_{off} + j·Q_{off})
    - *IQ_Rotation*: cx = cx·(cos(R)+j·sin(R))
    - *GainImbalance* and *PhaseImbalance*: let g = $10^{G/20}$. The I and Q envelopes of the signal I·cos(2πf_ct) − Q·sin(2πf_ct) can be recovered by multiplying it with cos(2πf_ct) and sin(2πf_ct) respectively and filtering the generated products at 2·f_c. When these imbalances are applied, the local oscillator for the Q channel is not sin(2πf_ct) but g·sin(2πf_ct + φ). The result is that the recovered value for the Q envelope is g·(Q·cos(φ) − I·sin(φ)).
8. If *MirrorSignal* = YES, then cx = conjugate(cx)
9. Finally, the output is calculated as follows (in the following equations, I(t) and Q(t) represent the I and Q envelopes of the signal after all the transformations and impairments described above have been applied):
    - When *OutputType* is set to *I/Q*,
      $output1 = S_a \cdot I(t)$
      $output2 = S_a \cdot Q(t)$
    - When *OutputType* is set to *Amp/Phase*,
      $output1 = S_a \cdot \sqrt{I^2(t) + Q^2(t)}$
      $output2 = S_p \cdot \frac{180}{\pi} \cdot \arctan \frac{Q(t)}{I(t)}$
    - When *OutputType* is set to *Amp/Frequency*,
      $output1 = S_a \cdot \sqrt{I^2(t) + Q^2(t)}$
      $output1 = \frac{S_f}{2\pi} \cdot \frac{d}{dt}\theta(t), \text{ where } \theta(t) = arctan\frac{Q(t)}{I(t)}$
10. See also: *Modulator* (algorithm)

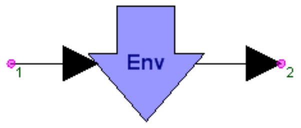
DownSampleEnv Part

Categories: *Analog/RF* (algorithm), *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
DownSampleEnv (algorithm)	Down Sampler for Envelope Signal

DownSampleEnv



Description: Down Sampler for Envelope Signal
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *DownSampleEnv Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Factor	Downsampling ratio	2		Integer	NO	[1:∞)
Phase	Downsampling phase	0		Integer	NO	[0:Factor-1]
AntiAliasingFilter	Turn off/on anti-aliasing filter before downsampling: OFF, ON	OFF		Enumeration	NO	
ExcessBW	Excess bandwidth of raised cosine anti-aliasing filter	0.5		Float	NO	[0:1]

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	envelope	NO

⚠ To avoid confusion on a schematic, it is best to **mirror** (flip) this symbol (instead of **rotating** it) when you need to have the input on the *right* and the output on the *left* of the symbol; otherwise the symbol arrow will point in the "wrong" direction. Select the part and press F6 to flip the symbol into the correct orientation.

Notes/Equations

1. This component downsamples an input timed signal to produce an output timed signal sampled with a time step that is Factor times the input time step.
2. This model reads *Factor* samples from the input and writes 1 sample to the output.
3. The *Phase* parameter identifies which one of the *Factor* input sample is to be used as the output sample.
   Let N1 = the Nth input sample (starting at count of zero).
   Then, TIn = TStartIn + N1*TStepIn where TStartIn is the time stamp for the first sample into this model.
   Let N2 = the Nth output sample (starting at count of zero) with one output sample occuring for every Factor input samples.
   Then, TOut = TStartOut + N2*TStepOut where TStartOut = TStartIn + Phase*TStepIn and TStepOut = Factor * TStepIn
   v_2 (TOut) = v_1 (TOut)
4. The AntiAliasingFilter parameter can be used to activate/de-activate an anti-aliasing filter before downsampling. When the input signal is baseband, the anti-aliasing filter is a lowpass raised-cosine filter with bandwidth equal to 1/(2 × TStepOut). When the input signal is a complex envelope signal, the anti-aliasing filter is a bandpass raised-cosine filter with bandwidth equal to 1/TStepOut. In both cases the filter has 20 × Factor taps (it introduces a delay of 10 × TStepOut) and its excess bandwidth can be set by the ExcessBW parameter.
5. To downsample a signal by a non-integer factor, a cascade of an upsampler and downsampler is needed. For example, to change the sampling rate of a signal from 73MHz (TStep = 13.69863 nsec) to 40MHz (TStep = 25 nsec), first pass it through the UpSampleEnv model(Faotor=40, Type=PolyPhaseFilter or Linear) and then through the DownSampleEnv model(Factor=73). To improve simulation speed, make sure the two Factors are relatively prime; if not, divide them with their GCD (greatest common divisor). For example, if in the previous case the original sampling rate was

72 MHz, the Factor of the UpSampleEnv model can be set to 5 (=40/8) and the Factor of the DownSampleEnb component can be set to 9 (=72/8); GCD (72, 40) = 8.

See:
DownSample (algorithm)
DownSampleVarPhase (algorithm)
UpSampleEnv (algorithm)

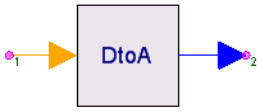
DtoA Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
DtoA (algorithm)	Digital to Analog Converter with Integral and Differential Nonlinearities

DtoA



Description: Digital to Analog Converter with Integral and Differential Nonlinearities
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *DtoA Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NBits	Number of bits	8		Integer	NO	[2:∞)
VRef	Reference voltage for output analog signal A: -VRef<=A<=VRef	1.0	V	Float	NO	(0:∞)
INL	Integral nonlinearity relative to least significant bit (LSB)	0.0		Float	NO	[DNL/2:∞)
DNL	Differential nonlinearity relative to least significant bit (LSB)	0.0		Float	NO	[0:∞)
InputDigitalFormat	Input digital format: Offset binary, Twos-complement	Offset binary		Enumeration	NO	
RepeatOutput	Output upsampling repeat ratio	1		Integer	NO	[1:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	D	input to D/A	int	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	A	output of D/A	real	NO

Notes/Equations

1. This component models a digital-to-analog converter with integral and differential nonlinearities. The input is a digital word in integer form. The output is a quantized real baseband signal.

   Note
   Bit-to-integer conversion is not performed within this model. For this conversion, place an external BitsToInt converter before this component.

2. This block reads 1 sample from the input and writes *RepeatOutput* samples to the output.
3. Summary of operation
   The *NBits*, *VRef*, *INL*, *DNL*, and *InputDigitalFormat* values are used to quantize the output *A* from the digital integer *D* input word. When *InputDigitalFormat* is *OffsetBinary*, then *D* is limited to the range [0, 2^ *NBits* - 1]. When *InputDigitalFormat* is *Twos-complement*, then *D* is limited to the range [-2^(*NBits* - 1), 2^(*NBits* - 1) - 1]. The output *A* is limited to values within the range of [- *VRef*, *VRef*]. The *INL* and *DNL* value affect the quantization levels as described in the following notes.
4. DNL (differential nonlinearity) error is defined as the difference between an actual output step width and the ideal value of 1 LSB (least significant bit, 1 LSB = 2 × VRef / 2 ^{NBits}). For an ideal digital-to-analog converter, in which the DNL=0 LSB, each output analog step equals 1 LSB. The DNL parameter is used to set the maximum value of DNLs. A DNL error specification of less than or equal to 1 LSB guarantees a monotonic transfer function.

   Note
   There is no guarantee that the DNL parameter value will be reached. The DNL error is modeled by a normal (Gaussian) distribution. The distribution has an approximate 1% probability that the DNL error will be equal to or greater than the DNL parameter value (or less than -DNL). Those numbers are then truncated to DNL (or -DNL) before further processing.

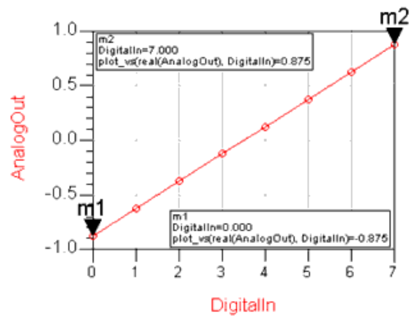
5. INL (integral nonlinearity) error is defined as the deviation (in LSB) of the digital-to-

analog converter actual transfer function from an ideal straight line. The INL parameter is used to set the maximum value of INLs.

Note
There is no guarantee that the INL parameter value will be reached; it depends on the NBits, INL, and DNL values. With larger NBits, smaller INL, and larger DNL, the INL will be reached more easily and frequently.

6. For an ideal DAC (INL=0 and DNL=0) the *A* output is in the range of [-(VRef - 0.5 LSB), (VRef - 0.5 LSB)] with value {-VRef + (i - 0.5) × LSB}, where i = 1, ... , 2 ^{NBits} .
7. AnalogOut vs. DigitalIn shows the output of an ideal DAC (INL=0 and DNL=0) with NBits=3, VRef=1, and RepeatOutput=1.
   In this case, the LSB = 0.25; and the output signal A is {-0.875, -0.625, -0.375, -0.125, 0.125, 0.375, 0.625, 0.875}.

AnalogOut vs. DigitalIn



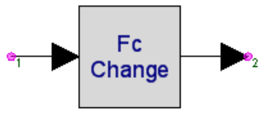
EnvFcChange Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
EnvFcChange (algorithm)	Envelope Signal Characterization Frequency Converter

EnvFcChange (Complex Envelope Signal Characterization Frequency Change)



Description: Envelope Signal Characterization Frequency Converter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *EnvFcChange Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
OutputFc	New characterization frequency	0	Hz	Float	NO	[0:∞)
Bandwidth	Bandwidth of bandpass filter centered at OutputFc (used when input fc=0)	0	Hz	Float	NO	[0:∞)†

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	envelope	NO

Notes/Equations

1. The EnvFcChange block converts a complex envelope signal with a defined characterization frequency to an equivalent complex envelope signal defined at another characterization frequency.
2. This block reads 1 sample from the *input* and writes 1 sample to the *output*.
3. EnvFcChange does not change the information content of the input signal. Only the signal representation is changed.
4. The signals at pin k, where k is 1 or 2, have complex envelope representation. $v_k = v_k(t)\exp(j2\pi f_k t)$, where $v_k(t) = v_{i_k}(t) + jv_{q_k}(t)$
5. When the input complex envelope signal is converted from its characterization frequency f_1 to the output characterization frequency f_2, the conversion algorithm transforms the input envelope value to an output envelope value at time t.
6. f_2 is set from *OutputFc*. Two cases for f_1 are considered.
   - **Case 1:** $f_1 > 0$
     - $v_2(t) = v_1(t)\exp(j2\pi(f_1 - f_2)t)$
     - if f_2 is zero, then the imaginary part of $v_2(t)$ is set to zero.
   - **Case 2:** $f_1 = 0$

     The assumption is made that the input is a bandpass signal with no significant energy at 0 Hz and that the energy of interest is located at $f_2 \pm$(*Bandwidth*/2).

     The output inphase and quadrature are extracted by multiplying the real input by $\cos(2\pi f_2 t)$ and by $\sin(2\pi f_2 t)$ and then low pass filtering the products with bandwidth *Bandwidth*.
     When the value of *Bandwidth* is 0, a default value of f_2 is used. Note that delay is introduced, because of the low pass filtering.
7. A warning message will be displayed when the input sample rate is too small to make the transformation without loss of information.

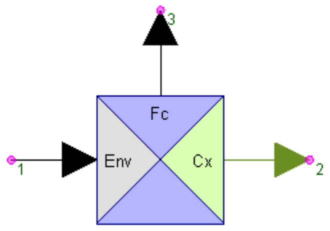
EnvToCx Part

Categories: *Analog/RF* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
EnvToCx (algorithm)	Envelope to Complex

EnvToCx



Description: Envelope to Complex
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *EnvToCx Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	complex	NO
3	fc	envelope	NO

Notes/Equations

1. EnvToCx decomposes *input* into a complex envelope and its characteristic frequency.
2. For every input sample, one sample to written to both outputs.
3. If *input* is a real baseband signal (v), then the *output* is real and set to the input value (v), and *fc* is a complex envelope signal set to 0+j*0 with a zero characteristic frequency.
4. If *input* is a complex envelope signal (i+j*q with non-zero characterization frequency f1), then the *output* is a complex envelope set to the input value (i+j*q with non-zero characterization frequency f1), and *fc* is a complex envelope signal set to 0+j*0 with non-zero characteristic frequency set to f1.

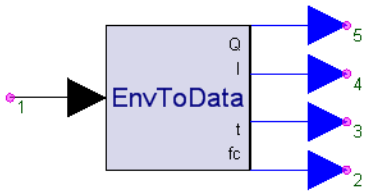
EnvToData Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
EnvToData (algorithm)	Convert Envelope into its Characteristic Frequency, Time, Inphase and Quadrature Values

EnvToData



Description: Convert Envelope into its Characteristic Frequency, Time, Inphase and Quadrature Values
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *EnvToData Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

Output Ports

Port	Name	Signal Type	Optional
2	fc	real	NO
3	time	real	NO
4	I	real	NO
5	Q	real	NO

Notes/Equations

1. EnvToData decomposes a complex envelope into its constituents.
2. For every sample read, one sample is written to all outputs.
3. If the input is a real baseband signal, only I and time are meaningful. Zeroes are output for Q and f $_c$.
4. Otherwise, the input complex envelope signal is decomposed into its constituent values of f $_c$, time, I and Q which are related to the input signal follows:
   - $V_{in}(time) = Re\{(I(time) + jQ(time))\exp(j2\pi f_c \cdot time)\}$

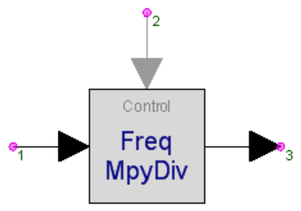
FreqMpyDiv Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
FreqMpyDiv (algorithm)	Envelope Signal Frequency Multiplier or Divider

FreqMpyDiv



Description: Envelope Signal Frequency Multiplier or Divider
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *FreqMpyDiv Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
MultDiv	Either frequency multiplier or freqency divider: Frequency multiplier, Frequency divider	Frequency multiplier		Enumeration	NO	
NominalX	Nominal frequency multiplication/division factor	1		Float	NO	(0:∞)
MaxX	Maximum limit for frequency multiplication/division factor (used when the optional control input is used)	2		Float	NO	(0:∞)
MinX	Minimum limit for frequency multiplication/division factor (used when the optional control input is used)	0.5		Float	NO	(0:∞)
OperatorType	Multiplier/divider type: Full signal, Phase only	Phase only		Enumeration	NO	

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	envelope	NO
2	control	input normalized optional control signal	envelope	YES

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output signal	envelope	NO

Notes/Equations

1. FreqMpyDiv is a frequency multiplier or a frequency divider that operates on a complex envelope depending on parameter MultDiv.
2. This model reads 1 sample from the inputs and writes 1 sample to the output.
3. The input signal must be a complex envelope signal with characterization frequency greater than zero. Real input signals are not allowed by this model. If use with a real signal is needed, then the input to this model can be preceeded with an *EnvFcChange* (algorithm) model that will recharacterize a real signal to its representation at a specified frequency.
4. Let X be the multiplier or divider value. When there is no control input, then X = *NominalX*. When there is a *control* input, then X = NominalX + *control*, with X limited in this case to MinX and MaxX.
5. Frequency multiplication is accomplished by first passing the input signal through a nonlinearity which raises the input signal to the power X and then passing the signal through a bandpass filter centered at $X \cdot f_{c_1}$, where f_{c_1} is the carrier frequency of the input. Computational efficiency is accomplished by directly calculating the output of the bandpass filter without explicitly performing the filtering operation.
   - Assuming the input signal is
     $V_1(t) = \Re\left((v_{I_1}(t) + jv_{Q_1}(t)) \exp(j2\pi f_{c_1} t)\right)$
   - The output signal is given by
     $V_2(t) = \Re\left(v_{2_{mag}}(\cos(v_{2_{ph}}) + j\sin(v_{2_{ph}})) \exp(j2\pi X f_{c_1} t)\right)$

where

$v_{2_{mag}} = \left(\sqrt{(v_{I_1}^2 + v_{Q_1}^2}\right)^a$

a=1 when Type is *RF phase only*
a=X when Type is *full signal*
and

$v_{ph} = X \cdot atan2(v_{Q_1}, v_{I_1})$

6. Frequency division with divisor X is a frequency multiplication operation with 1/X.

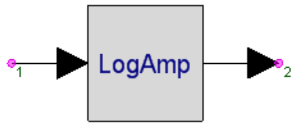
LogAmp Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
LogAmp (algorithm)	Logarithmic Amplifier

LogAmp



Description: Logarithmic Amplifier
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *LogAmp Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Sensitivity	Log sensitivity in volts/dB	0.1	V	Float	NO	(0:∞)
PMin	Minimum input power in dBm for logarithmic amplification	-80		Float	NO	(-200:∞)
E	Peak log error in dB	0		Float	NO	[0:∞)
Ec	Log error cycle in dB	0		Float	NO	(0:∞)
RefR	Reference resistance	50	ohm	Float	NO	(0:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	envelope	NO

Notes/Equations

1. LogAmp models a logarithmic amplifier with some non-ideal behavior.
2. This model reads 1 sample from the input and writes 1 sample to the output.
3. The basic equation describing this amplifier is
   $V_2(t) = K\log(V_1(t)/V_L) + \epsilon$
   where
   K determines the logarithmic slope
   V_L determines the minimum input voltage required for logarithmic amplification
   ε is the deviation of the amplifier from the ideal.
4. The following equations describe the algorithm used for this model.
   - Let
     $V_1(t) = \Re\left((v_{I_1}(t) + jv_{Q_1}(t))\exp(j\omega_c t)\right.$
     $A(t) = \sqrt{v_{I_1}^2(t) + v_{Q_1}^2(t)}$, envelope of the input signal
     $P_A = 10\log(\frac{A^2(t)}{2\cdot RefR}) + 30$, the power of the envelope in dBm
     $\epsilon = Sensitivity \cdot E \cdot \sin(2\pi\frac{P_A - PMin}{Ec})$
     $V_L = \sqrt{2}\left(RefR \cdot 10^{\frac{PMin-30}{10}}\right)^{0.5}$
     $$M(t) = \begin{cases} 20 \cdot Sensitivity \cdot \log\left(\frac{A(t)}{V_L}\right) + \epsilon & \text{if } A(t) > V_L \\ 0 & \text{otherwise} \end{cases}$$
   - Then, the output signal *V* $_2$ *(t)* is given by the equation
     $V_2(t) = \frac{M(t)}{A(t)}V_1(t)$

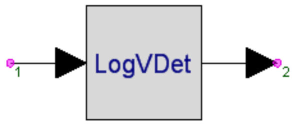
LogVDet Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
LogVDet (algorithm)	Logarithmic Video Detector

LogVDet



Description: Logarithmic Video Detector
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *LogVDet Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Sensitivity	Log sensitivity in volts/dB	0.1	V	Float	NO	(0:∞)
PMin	Minimum input power in dBm for logarithmic amplification	-80		Float	NO	(-200:∞)
E	Peak log error in dB	0		Float	NO	[0:∞)
Ec	Log error cycle in dB	0		Float	NO	(0:∞)
RefR	Reference resistance	50	ohm	Float	NO	(0:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	envelope	NO

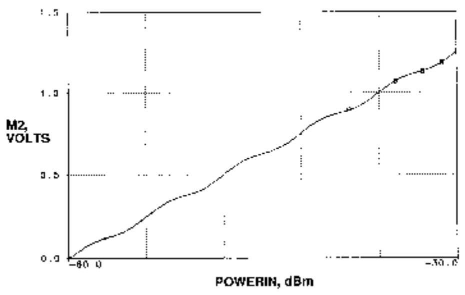
Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	envelope	NO

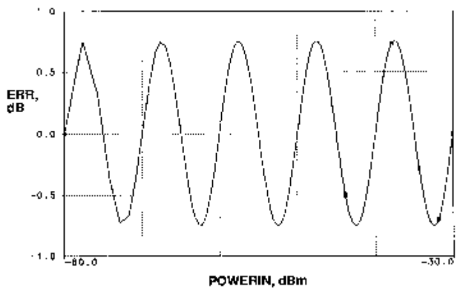
Notes/Equations

1. LogVDet models a logarithmic video detector with some non-ideal behavior.
2. This model reads 1 sample from the input and writes 1 sample to the output.
3. The basic equation describing this detector is:
   $$V_2(t) = K \log\left(\frac{V_1(t)}{V_L}\right) + \epsilon$$
   where
   K determines the logarithmic slope
   V_L determines the minimum input voltage required for logarithmic video detection
   ε is the deviation of the detector from the ideal.
4. The following equations describe the algorithm used for this model.
   - Let
     $$V_1(t) = \Re\left\{\left(v_{I_1}(t) + jv_{Q_1}(t)\right)\exp\left(j\omega_c t\right)\right\}$$
     $$A(t) = \sqrt{v_{I_1}^2(t) + v_{Q_1}^2(t)}\text{ , envelope of input signal}$$
     $$P_A = 10\log\left(\frac{A^2(t)}{2\cdot RefR}\right) + 30\text{ , power of envelope in dbm}$$
     $$\epsilon = Sensitivity \cdot E \cdot \sin\left(2\pi\frac{P_A - PMin}{Ec}\right)$$
     $$V_L = \sqrt{2}\left(RefR \cdot 10^{\frac{PMin-30}{10}}\right)^{0.5}\text{ , voltage level corresponding to PMin}$$
   - Then, the output signal *V*$_2$*(t)* is given by the equation
     $$V_2(t) = \begin{cases} 20 \cdot Sensitivity \cdot \log\left(\frac{A(t)}{V_L}\right) + \epsilon & \text{if } A(t) > V_L \\ 0 & \text{otherwise} \end{cases}$$

     Example: S= 0.025v/dB, PMin = −80 dBm, E = 0.75 dB, EC = 10 dB



M2 vs. POWERIN



ERR vs. POWERIN

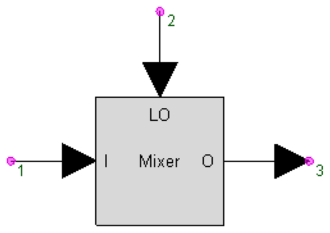
Mixer Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Mixer (algorithm)	Envelope Signal Mixer

Mixer (Signal Mixer)



Description: Envelope Signal Mixer
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Mixer Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
ConvGain	Conversion gain in dB	0		Float	NO	(-∞:∞)
EnableNoise	Enable mixer noise: NO, YES	YES		Enumeration	NO	
NoiseFigure	Double sideband noise figure in dB	0		Float	NO	[0:∞)
Sideband	Mixer primary output sideband: Lower, Upper	Lower		Enumeration	NO	
SidebandSuppression	Suppression of the output alternate sideband in dB	-200		Float	NO	(-∞:0)
RfRej	RF to outut rejection in dB	-200		Float	NO	(-∞:0)
LoRej	LO to output rejection in dB	-200		Float	NO	(-∞:0)
LoRfIso	LO to RF isolation in dB	-200		Float	NO	(-∞:0)
RfLoIso	RF to LO isolation in dB	-200		Float	NO	(-∞:0)
SOIout	Output second order intercept power	1.0e17	W	Float	NO	(0:∞)†
TOIout	Output third order intercept power	1.0e17	W	Float	NO	(0:∞)†
RefR	Reference resistance	50	ohm	Float	NO	(0:∞)

Input Ports

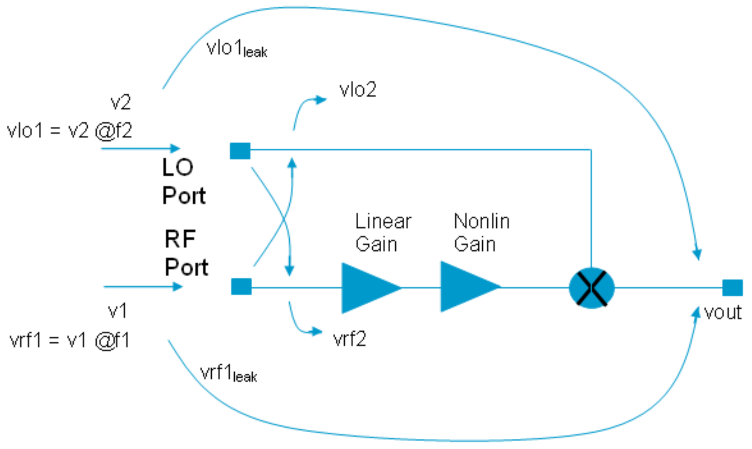
Port	Name	Description	Signal Type	Optional
1	input1	input signal 1	envelope	NO
2	input2	input signal 2	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output signal	envelope	NO

Notes/Equations

1. Mixer models an RF mixer for use with complex envelope input and local oscillator (LO) and includes many non-ideal mixer features such as noise figure, RF and LO leakage products, and 2 nd and 3 rd order nonlinearities.
2. This block reads 1 sample from *input1* and *input2* and writes 1 sample to *output*.
3. The Mixer primary signal of interest at the *input1* is the RF signal with frequency f_{RF}. The RF signal mixes with the LO input at *input2* with frequency f_{LO} to produce the lower sideband output signal with frequency $|f_{LO} - f_{RF}|$ and the upper sideband output signal with frequency $(f_{LO} + f_{RF})$. The mixer is operated with selection of either the lower or upper output sideband as the primary sideband. The mixer conversion gain, *ConvGain*, is the ratio of the desired output sideband to the RF input signal level. The other sideband output level is controlled by the *SidebandSuppression* value.
4. The Mixer block internal signal flow is diagramed.



- Mixer operates with LO limiting and does not support any starved LO conditions, i.e. the LO always drives the mixer into saturation.
- Note that the linear gain precedes the nonlinear gain which precedes the mixing process.
- The mixing process is a 'linear' process performing frequency translation that results in lower and upper sideband products.
- vrf1 is the primary RF input signal at frequency f1 including any noise from *NoiseFigure*.
- vrf2 is the secondary RF input signal leakage due to the LO at frequency f2 appearing at the RF input.
- vlo1 is the primary LO input signal at frequency f2.
- vlo2 is the secondary LO input signal due to leakage from the RF input at frequency f1 appearing at the LO input.

5. The Mixer input RF image frequency, $f_{RFImage}$, is defined as the other input frequency which results in a mixing product output at the specified output frequency. The input image is typically only of interest for receiver/down-conversion applications and is given by $f_{RFImage} = 2 * f_{LO} - f_{RF}$.
   The frequencies, f_{RF} and $f_{RFImage}$, both mix down with f_{LO} to generate the output at $|f_{LO} - f_{RF}|$. The signals f_{RF} and $f_{RFImage}$ are called images of each other with respect to the f_{LO}.
   In typical simulations, the $f_{RFImage}$ is often input noise that is located at a frequency away from the desired f_{RF} input modulated signal.
6. The Mixer output operating without nonlinearity can be defined with this expression:
   $vout = vout_A + vout_B + vout2_A + vout2_B + vlo1_{leak} + vrf1_{leak}$
   where
   - $vout_A$ = mixer output due to mixing with the primary LO signal, vlo1, and at the primary sideband based on the parameters *Sideband* and *ConvGain*.
   - $vout_B$ = mixer output due to mixing with the primary LO signal, vlo1, and at the alternate sideband based on the suppression parameter, *SidebandSuppression*, and *ConvGain*.
   - $vout2_A$ = mixer output due to mixing with the secondary LO signal, vlo2, at the primary sideband based on the parameters *Sideband* and *ConvGain*.
   - $vout_B$ = mixer output due to mixing with the secondary LO signal, vlo2, at the alternate sideband based on the suppression parameter, *SidebandSuppression*, and *ConvGain*.
   - $vlo1_{leak}$ = mixer output due to leakage of the primary LO input signal appearing at the output based on the rejection parameter, *LoRej*.
   - $vrf1_{leak}$ = mixer output due to leakage of the primary RF input signal appearing at the output based on the rejection parameter, *RfRej*.
7. The Mixer output operating with nonlinearity is defined by the *SOIout* and *TOIout* parameters.
   For definition of the SOI and TOI nonlinearities, see the detail discussion in the documentation for the *AmplifierBB* (algorithm) block.
   The nonlinearity is applied only to the RF path before the mixing process which is composed of vrf1 and vrf2.
8. The Mixer self generated noise is added to the primary RF input signal, vrf1, and is defined by the parameters *NoiseFigure* and *RefR* .
   NoiseFigure for Mixer is defined as a double sideband noise figure, NFdsb, which follows industry conventions. The double sideband noise figure is converted to an equivalent single sideband noise figure, NFssb. That conversion is dependent on whether the lower or the upper sideband is selected.
   - In general, the NFssb is related to the NFdsb as follows

NFssb = NFdsb + 10 * log((G1 + G2 +...) / G1)
where
    - G1 is the primary power gain from the input noise frequency to the output noise frequency
    - G2 +... is the sum of all higher order mixing gains which is mixed from some input frequency to the output noise frequency.
- For a mixer without input image rejection as is the case for this Mixer model, G2 is significant and is often equal to G1 under small-signal operation while G3 +... is zero under small-signal conditions.
  Thus, the simplified expression, NFssb = NF + 10 * log(1 + G2 / G1), is used.
  Note that the Mixer self noise is considered to be broadband such that it exists at all frequencies.
- Consider the case when *Sideband* is _Lower and choose $f_{RF} < f_{LO}$.
  The primary lower output sideband is $|f_{LO} - f_{RF}|$. The input image for down-conversion is at $2 * f_{LO} - f_{RF}$. Its lower output sideband is due to mixing down by the f_{LO} which also gives $|f_{LO} - f_{RF}|$. The gain factor G2 is the same as G1 since both noise results from a frequency down conversion. Thus, the NFssb is 3 dB more than the NFdsb, i.e. NFssb is *NoiseFigure* + 3.010299.
- Consider the case when *Sideband* is *Upper* and choose $f_{RF} < f_{LO}$.
  The primary upper output sideband is $|f_{LO} + f_{RF}|$. The input image for up-conversion is at $2 * f_{LO} + f_{RF}$. Its lower output sideband is due to mixing down by the f_{LO} to also give $|f_{LO} + f_{RF}|$. At the same frequency, the first noise term (G1) is due to an up conversion and the second noise term (G2) is due to a down conversion. Thus, the mixer gain for G2 will be different from the gain for G1 due to *SidebandSuppression*. The single sideband noise figure is increased only if the alternate sideband is not suppressed, i.e. NFssb = NoiseFigure + 10 * log(1 + $10^{(\textit{SidebandSuppression} / 10)}$).

9. vn_rms, the rms noise voltage, is used to generate a complex Gaussian random value at each simulation time point which is then added to the input complex envelope.
   Let
   k = Boltzmann constant is 1.380622e-23 J/K
   sr = simulation sample rate of the block input
   t0 = reference temperature in Kelvin is set at 290 K
   nf = $10^{(\text{NFssb} / 10)}$
   then $v_{n_{rms}} = \sqrt{k \cdot (t0 \cdot (nf - 1)) \cdot sr \cdot RefR}$
10. How does the simulation sample rate impact the Mixer model. Let the simulation sample rate be called sr and the primary mixer output frequency be called f_{out}.
    - If *Sideband* is *Upper*, then f_{out} is $f_{LO} + f_{RF}$.
    - Else *Sideband* is *Lower*, then f_{out} is $|f_{LO} - f_{RF}|$ provided there is ideal LO to RF and RF to LO isolation, otherwise f_{out} is set to 0 Hz.
      There is ideal LO to RF isolation when *LoRfIso* is less then -200 dB. There is ideal RF to LO isolation when *RfLoIso* is less then -200 dB.
      When there is non-ideal LO to RF or RF to LO isolation, f_{out} is set to 0 Hz, since the primary artifact of non-ideal LO to RF and RF to LO is frequency content at or near 0 Hz.
    - For Mixer usage without suppression, leakage, isolation and nonlinear characteristics, the model is not dependent on the simulation sample rate. It simply mixes the RF input with the LO input to produce the selected sideband output with mixer noise if specified.
    - However, when the there are any suppression, leakage, isolation or nonlinear characteristics specified, then the simulation sample rate will impact how these additional effects are included in the mixer output signal.
      Each of these additional mixer characteristics may result in frequency terms that reside outside the frequency range [f_{out} - sr / 2, f_{out} + sr / 2]. If frequency terms are outside this range, they will be excluded from the output signal.
      In general, these frequency terms are excluded without warning. Warnings are issued when the sample rate is too small to support the effects of leakage terms when *LoRej* > -200 dB or when *SidebandSuppression* > -200 dB.

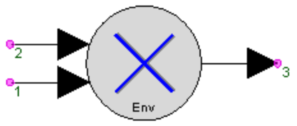
MpyEnv Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
MpyEnv (algorithm)	Two Input Envelope Multiplier

MpyEnv



Description: Two Input Envelope Multiplier
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *MpyEnv Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
SideBandType	Multiplication product type: Lower sideband, Upper sideband, Both sidebands	Upper sideband		Enumeration	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	input1	input signal 1	envelope	NO
2	input2	input signal 2	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output signal	envelope	NO

Notes/Equations

1. MpyEnv multiplies the two input complex envelope signals. It can be used to model an up-converter, down-converter, or double sideband modulator.
2. This model reads 1 sample from all inputs and writes 1 sample to the output.
3. The signals at pin k, where k is 1, 2, or 3, have complex envelope representation:
   $v_k = v_k(t)\exp(j2\pi f_k t)$, where $v_k(t) = v_{I_k}(t) + jv_{Q_k}(t)$
4. For SidebandType = *Both*
   - When both inputs are complex envelope: $v_3(t) = 0.5\, v_1(k)\, \mathrm{conj}(v_2(t)) + 0.5\, \mathrm{conj}(v_1(t))\, v_2(t)$; $f_3 = \max(f_1, f_2)$
   - When both inputs are real: $v_3(t) = v_1(t)\, v_2(t)$; $f_3 = 0$
   - When one input is real: $v_3(t) = v_1(t)\, v_2(t)$; $f_3 = \max(f_1, f_2)$
5. For SidebandType = *Upper sideband*
   - When either input is real: SidebandType is forced to *Both* and the output is set for that case.
   - When both inputs are complex envelope: $v_3(t) = 0.5\, v_1(t)\, v_2(t)$; $f_3 = f_1 + f_2$
6. For SidebandType = *Lower sideband*
   - When either input is real: SidebandType is forced to *Both* and the output is set for that case.
   - When both inputs are complex envelope and $f_1 > f_2$: $v_3(t) = 0.5\, v_1(t)\, \mathrm{conj}(v_2(t))$; $f_3 = f_1 - f_2$
   - When both inputs are complex envelope and $f_2 > f_1$: $v_3(t) = 0.5\, \mathrm{conj}(v_1(t))\, v_2(t)$; $f_3 = f_2 - f_1$
   - When both inputs are complex envelope and $f_1 = f_2$: $v_3(t) = 0.5\, v_1(t)\, \mathrm{conj}(v_2(t))$; $f_3 = 0$ and the imaginary part of $v_3(t)$ is set to 0.0.
7. In the above, the function conj() takes the conjugate of the complex argument.

See:
AddEnv (algorithm)
SubEnv (algorithm)

MpyMultiEnv (algorithm)

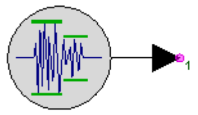
NoiseFMask Part

Categories: *Analog/RF* (algorithm), *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
NoiseFMask (algorithm)	Noise Generator with Frequency Domain Mask Specification

NoiseFMask



Description: Noise Generator with Frequency Domain Mask Specification
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *NoiseFMask Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
FCarrierOption	Signal carrier frequency option: Carrier at center of NoiseMask data, Defined by FCarrier	Carrier at center of NoiseMask data		Enumeration	NO		
FCarrier	Carrier frequency for the noise (used if FCarrierOption is Defined by FCarrier)	1000000	Hz	Float	NO	(0:∞)	
NoiseMask	Noise specification defined with pairs of values for frequency (Hz), noise level (dBm/Hz)			Floating point array	NO		
ResBW	Resolution frequency bandwidth for noise spectrum	1e6	Hz	Float	NO	(0:∞)†	
RefR	Reference resistance	50	ohm	Float	NO	(0:∞)	
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D

Output Ports

Port	Name	Description	Signal Type	Optional
1	output	output signal	envelope	NO

Notes/Equations

1. NoiseFMask generates an RF (complex envelope) noise signal with a power spectral density that follows the frequency domain noise mask as defined by *NoiseMask*.
2. The *NoiseMask* parameter is an array of values that defines the mask. The values are interpreted as pairs of frequency in Hz and noise power level in dBm/Hz.
3. The *FCarrierOption* and *FCarrier* parameters define the characterization frequency (fc) of the output signal.
   - For *FCarrierOption* set to *Carrier at center of NoiseMask data*, fc is the average of the effective smallest and largest frequencies listed in *NoiseMask*.
   - For *FCarrierOption* set to *Defined by FCarrier*, fc is *FCarrier*.
4. The output signal is modeled as a sum of tones.
   - Number of tones is (f_{max} - f_{min}) / ResBW. Let SR be the simulation sample rate for this model instance. If f_{min} is less than fc - (SR / 2), then it is reset to this limit. If f_{max} is greater than fc + (SR / 2), then it is reset to this limit. More tones allow for more accurate modeling in the output noise. However, using more tones will increase simulation time.
   - Frequency of the *i* th tone is selected from a uniform distribution in the interval [f + i × *ResBW*, f + (i + 1) × *ResBw*).

min min

- Amplitude of the *i* th tone is selected from a normal distribution centered around an amplitude that corresponds to the power level specified in the mask. Power levels for tones whose frequency is not listed in the *NoiseMask* parameter are interpolated. The power level is converted to voltage level using the reference resistance, *RefR*.
- Initial phase of each tone is selected from a uniform distribution in the interval [0, 2 × pi).

5. For other parameter descriptions, see *Timed Sources* (algorithm).

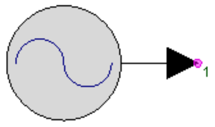
Oscillator Part

Categories: *Analog/RF* (algorithm), *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Oscillator (algorithm)	Oscillator with Carrier Frequency

Oscillator



Description: Oscillator with Carrier Frequency
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Oscillator Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
Frequency	RF tone frequency	1000000	Hz	Float	NO	(0:∞)	
Power	RF tone carrier power	.010	W	Float	NO	[0:∞)	
Phase	RF tone carrier phase	0.0	deg	Float	NO	(-∞:∞)	
RandomPhase	Set phase of RF tone to random value between -PI and +PI: NO, YES	NO		Enumeration	NO		
PhaseNoiseData	Phase noise specification - pairs of offset freq (Hz) and SSB phase noise level (dBc/Hz)			Floating point array	NO		
PN_Type	Phase noise model type with random or fixed offset freq spacing and amplitude: Random PN, Fixed freq offset, Fixed freq offset and amplitude	Random PN		Enumeration	NO		
NDensity	Noise spectral density added	0	W	Float	NO	[0:∞)	
RefR	Reference resistance	50	ohm	Float	NO	(0:∞)	
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D

Output Ports

Port	Name	Description	Signal Type	Optional
1	output	output signal	envelope	NO

Notes/Equations

1. Oscillator generates an RF (complex envelope) defined by its carrier frequency and with optional thermal noise and phase noise.
2. The frequency, power, and phase of the first tone are defined by the *Frequency*, *Power* and *Phase* parameters, respectively.
3. If RandomPhase=Yes, then the *Phase* is ignored and a random phase between −180 and +180 degrees is used.
4. Noise power added to the output signal is NDensity_W × BW where
   NDensity_W is the noise density in Watts/Hz. The *NDensity* parameter specified in any power unit automatically converts to NDensity_W.
   BW is the simulation bandwidth which, for complex envelope signal, is the sample rate.
   The rms voltage associated with the noise power is $\sqrt{NDensity\_W \cdot BW \cdot RefR}$

.
The value of *NDensity* (in Watts/Hz) is related to temperature in Kelvin as kT where k is the Boltzmann constant (1.3806504e-23). At the standard system temperature of 290 Kelvin (16.85 Celsius), the *NDensity* is 4.00388587e-21 Watts/Hz (-173.975 dBm/Hz).

5. A phase noise characteristic is specified in the PhaseNoiseData list. This list contains double values of offset frequency (Hz) and single sideband relative power level (dBc/Hz).

```
PhaseNoiseData = "[100, -50, 1000, -70, 1e5, -90]" or alternatively can reference an array
defined using Math Language in an Equations sheet.
```

The phase noise characteristic defined from this list describes a frequency domain specification for phase noise. Interpolation is applied between these frequency domain points as needed to give a full time domain simulation definition for this phase noise.

Phase noise is modeled as a sum of tones that modulates the phase of the main tone. Let $f_{offsetMax}$ ($f_{offsetMin}$) be the maximum (minimum) offset frequency specified in PhaseNoiseData. Then the number of tones N used to model phase noise is equal to $f_{offsetMax}$ / $f_{offsetMin}$.
In the following, let ROut = RefR, the power specification reference resistance.
Let P_i be the phase noise power at frequency offset f_i. The phase (φ_i), frequency (f_i), and amplitude (or modulation index) (η_i) of each tone is given by:

When PN_Type = Fixed freq offset and amplitude

φ_i is a random variable uniformly distributed in [0, 2π)

$$f_i = i \times f_{offsetMin}$$

$$\eta_i = \sqrt{2 \times P_i \times ROut}$$

When PN_Type = Fixed freq offset

φ_i is a random variable uniformly distributed in [0, 2π)

$$f_i = i \times f_{offsetMin}$$

η_i is a random variable with a Rayleigh distribution and mean value

$$\sqrt{2 \times P_i \times ROut}$$

When PN_Type = Random PN

φ_i is a random variable uniformly distributed in [0, 2π)

f_i is a random variable uniformly distributed in
$$[(i - 0.5) \times f_{offsetMin}, (i + 0.5) \times f_{offsetMin})$$

η_i is a random variable with a Rayleigh distribution and mean value

$$\sqrt{2 \times P_i \times ROut}$$

For the first two cases *Fixed freq offset and amplitude* and *Fixed freq offset*, phase noise is a sum of tones whose frequencies are integer multiples of the same frequency ($f_{offsetMin}$). Therefore, phase noise will be periodic with period 1 / $f_{offsetMin}$ and all the phase noise signal power will be located at the discrete frequencies that are integer multiples of $f_{offsetMin}$. When a spectrum analysis is performed on this signal and the resolution bandwidth is equal to $f_{offsetMin}$ / M, where M is an integer, the spectrum will have spectral nulls (e.g. -250 dBm) at all frequencies that are not an integer multiple of $f_{offsetMin}$. The integrated power in a bandwidth of $f_{offsetMin}$ will still be what one expects based on the phase noise data specification but it will all be concentrated at one frequency (the one that is an integer multiple of $f_{offsetMin}$).

For better phase noise modeling it is recommended that simulations be performed with PN_Type set to Random PN (default). The other values for PN_Type can be used to

demonstrate/understand the phase noise modeling algorithm and are not recommended for use in practical simulations.
The single sideband phase noise in dBc/Hz is

$$\Im(f_i) = 10 \times \log\left(\frac{\eta_{rms}^{2}(f_i)}{2}\right) \text{ at offset frequency } f_i$$

where
$\eta_{rms}(f_i)$ = root mean square of modulation index η_i at offset frequency f_i.
The modulation index is related to the signal power and single sideband phase noise power as follows:

$$\frac{P_{ssb}(f_i)}{P_c} = \frac{\eta_{rms}^{2}(f_i)}{2}$$

where
P_c = signal power
$P_{ssb}(f_i)$ = signal sideband power at offset frequency *f* i
Therefore, single sideband phase noise in dBc/Hz can also be expressed as:

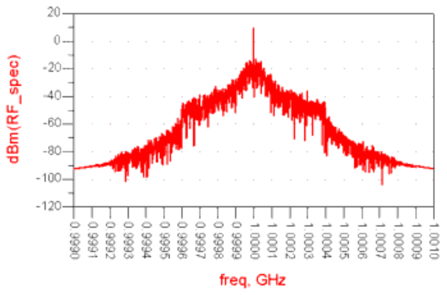
$$\Im(f_i) \text{ in dBc/Hz} = P_{ssb}(f_i)dBm - ResBWdB - P_c dBm$$

where
$P_{ssb}(f_i)dBm$ = simulated single sideband phase noise power per simulation frequency resolution bandwidth ResBW
ResBWdB = factor for frequency resolution bandwidth ResBW used during simulation = 10 × log10(ResBW)
P_c *dBm* = signal power in dBm

Note
If the phase noise settings result in the summed phase noise higher than *tone power* (dBm) - 10 dB, a warning message will be generated:
Phase noise violates small signal modeling requirement; phase noise power exceeds Tone_Power (dBm) - 10 dB.
However, the simulation will continue with the current parameter settings.

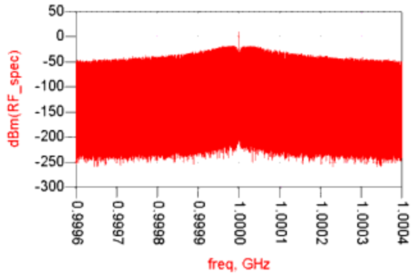
A phase noise modeling example is demonstrated here. Consider the simple design with an Oscillator used to generate a 1 GHz tone at a power level of 10 dBm with 50 Ohms reference resistance. The tone is colored with phase noise, whose frequency specification is defined with PhaseNoiseData = 1.0e3, -70, 1.0e4, -60, 4.0e4, -60, 4.0e5, -90. For this example, $f_{offsetMin}$ is 1 kHz and $f_{offsetMax}$ is 400 kHz. The simulation sample rate (Sample_Rate) is set to (8192 / Stop_Time) Hz, which is large enough to resolve the maximum phase noise frequency offset data point at 400 kHz. The simulation stop time (Stop_Time) is set to 1 msec, which is large enough to resolve the lowest phase noise frequency offset data point at 1 kHz.
The spectrum of the signal at the output of N_Tones is measured using a SpectrumAnalyzer component. The spectrum is shown in Signal Spectrum with PN_Type=Random PN and TStop=1 msec. The resolution bandwidth of the spectrum measurement is 1 / (Stop - Start) = 1 / (1 msec) = 1 kHz. This means that each spectral tone displayed will be at multiples of 1 kHz from the carrier frequency of 1 GHz. For noise power integrated over a 1 kHz bandwidth, the power would be 30 dB (=10 × log10(1000)) more than that in a 1 Hz bandwidth.

Signal Spectrum with PN_Type= *Random PN* and TStop=1 msec



Signal Spectrum with PN_Type=Fixed freq offset and amplitude and TStop=2 msec shows the signal spectrum when PN_Type= *Fixed freq offset and amplitude* and TStop=2 msec. The resolution bandwidth of the spectrum measurement is 1 / (Stop - Start) = 0.5 kHz. As explained in note 6, all phase noise signal power will be located at frequencies that are integer multiples of $f_{offsetMin}$ = 1 kHz and the spectrum values at frequencies (M+0.5) kHz (where M is an integer) will be practically 0.

Signal Spectrum with PN_Type= *Fixed freq offset and amplitude* and TStop=2 msec

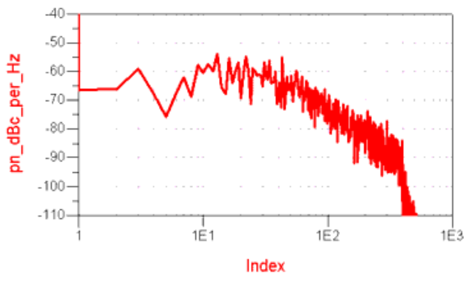


To view the phase noise spectrum versus spectral tone offset index, a signal processing network can be created to perform the FFT on the collected RF complex time domain waveform. Summed powers in the upper and lower sidebands are averaged and results are converted into dBm to obtain the single sideband phase noise power in dBm per simulation frequency resolution bandwidth versus offset spectral tone. This resultant single sideband phase noise spectrum can be displayed versus spectral tone offset index as shown in Phase Noise Spectrum with PN_Type=Random PN.

In Phase Noise Spectrum with PN_Type=Random PN, 1 kHz offset occurs at index 1, 10 kHz offset occurs at index 10, 40 kHz offset occurs at index 40, and 400 kHz offset occurs at index 400. As can be seen, this figure agrees with the PhaseNoiseData specified.
The phase noise data displayed in Phase Noise Spectrum with PN_Type=Random PN was generated using PN_Type= *Random PN* . In Phase Noise Spectrum with PN_Type=Fixed freq offset and amplitude, phase noise is displayed with PN_Type= *Fixed freq offset and amplitude* . As can be seen, Phase Noise Spectrum with PN_Type=Fixed freq offset and amplitude agrees much better (compared to Phase Noise Spectrum with PN_Type=Random PN) with the PhaseNoiseData specified, since in this case there is no randomness in the values of f_i and η_i .

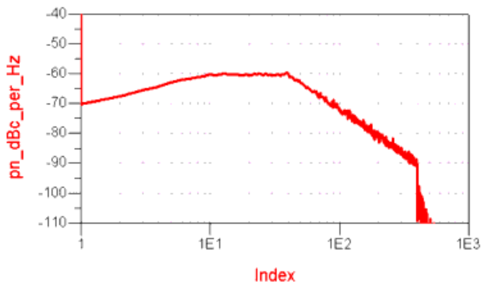
The results of Phase Noise Spectrum with PN_Type=Random PN and Phase Noise Spectrum with PN_Type=Fixed freq offset and amplitude were obtained by setting the FFT size to 2^{13}.

Phase Noise Spectrum with PN_Type= *Random PN*



Eqn pn_dBc_per_Hz = pn_spec_dBm - 10 - 30

Phase Noise Spectrum with PN_Type= *Fixed freq offset and amplitude*



Eqn pn_dBc_per_Hz = pn_spec_dBm - 10 - 30

For other parameter descriptions, see Time Burst Sources.

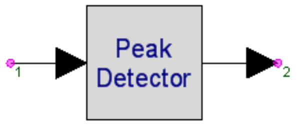
PeakDetector Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
PeakDetector (algorithm)	Peak Detector

PeakDetector



Description: Peak Detector
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *PeakDetector Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
ChargeTimeConstant	Output voltage charge time constant	0	s	Float	NO	[0:∞)
DecayTimeConstant	Output voltage decay time constant	20e-6	s	Float	NO	[0:∞)
VThreshold	Voltage threshold for detection	0	V	Float	NO	(-∞:∞)
VTransWidth	Voltage transition width	0	V	Float	NO	[0:∞)
Polarity	Polarity of the peak detector: positive, negative	positive		Enumeration	NO	

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	envelope	NO

Notes/Equations

1. PeakDetector captures the previous peak input and includes charge and decay time constants.
2. This model reads 1 sample from the input and writes 1 sample to the outputs.
3. Define the signal to be detected as vs. When the input is a real baseband signal, let vs be the input signal, vs = v1. When the input is a complex envelope signal, and *Polarity = positive* , let vs be the magnitude of the input signal, vs = |v1|. Otherwise, let vs be the negative of the input signal, vs = -|v1|.
4. The model operation can be defined with an input signal detection process and a signal output process.
   - The detection process is dependant on the *Polarity*, *VThreshold* and *VTransWidth* values and results in the detected value vdet.
   - The output process is dependant on the detected value, vdet, and the *ChargeTimeConstant* and *DecayTimeConstant* and results in the output value vout.
5. Input signal detection process for vdet when *Polarity = positive*
   - Require: *VThreshold* >= 0.
   - When vs < *VThreshold - VTransWidth* / 2, vdet = 0.
   - When *VTransWidth* = 0 and vs >= *VThreshold* level, the peak detection is instantaneous and vdet is:
     . $\mathrm{vdet} = \mathrm{vs} - \mathrm{VThreshold}$
   - When *VTransWidth* > 0 and vs >= *VThreshold - VTransWidth* / 2, then [*VThreshold - VTransWidth* / 2, *VThreshold + VTransWidth* / 2] defines a transition range for vs over which the detection is weighted and vdet is:
     . $vdet = \frac{0.5}{VTransWidth} \cdot (vs - (VThreshold - \frac{VTransWidth}{2}))^2$
6. Input signal detection process for vdet when *Polarity = negative*
   - Require: *VThreshold* <= 0.
   - When the vs > *VThreshold + VTransWidth* / 2, vdet = 0.
   - When *VTransWidth* = 0 and vs <= *VThreshold* level, the peak detection is instantaneous and vdet is:
     . $\mathrm{vdet} = \mathrm{vs} - \mathrm{VThreshold}$

- When *VTransWidth* > 0 and vs <= *VThreshold* + *VTransWidth* / 2, then [*VThreshold* - *VTransWidth* / 2, *VThreshold* + *VTransWidth* / 2] defines a transition range for vs over which the detection is weighted and vdet is:
  - $vdet = \frac{0.5}{VTransWidth} \cdot (vs - (VThreshold + \frac{VTransWidth}{2}))^2$

7. Signal output process for vout
   - When the new value for vdet is greater then the prior value for vout, then the new vout value will charge to this new vdet value dependant on the *ChargeTimeConstant*.
     - When *ChargeTimeConstant* = 0 and vdet > vout, then the charge is instantenous and vout = vdet.
     - When *ChargeTimeConstant* > = 0, then a test is performed to determine if vout should continue to charge.
       - $vtest = vout + (vdet - vout) \cdot (1. - exp(-1/ChargeTimeConstant/SR))$

       Where, SR = the simulation sample rate for the model instance.
       If vtest > vout, then vout is reset to vtest.
   - When the new value for vdet is less then the prior value for vout, then the new vout value will decay to this new vdet value dependant on the *DelayTimeConstant*.
     - When *DecayTimeConstant* = 0 and vdet < vout, then the decay is instantaneous and vout = vdet.
     - When *DecayTimeConstant* is large, then vout is held indefintely until changed by a new vdet value.
     - When *DecayTimeConstant* > = 0, then a test is performed to determine if vout should continue to decay.
       - $vtest = vout * exp(-1/DecayTimeConstant/SR)$

       Where, SR = the simulation sample rate for the model instance.
       If vtest < vout, then vout is reset to vtest.

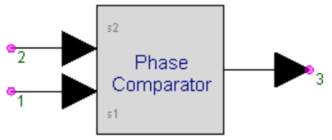
PhaseComparator Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
PhaseComparator (algorithm)	Phase Comparator

PhaseComparator



Description: Phase Comparator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *PhaseComparator Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
PhaseCharacteristicType	Type of analog phase comparator: PhaseFreq, Sinusoidal, Triangular	PhaseFreq		Enumeration	NO	
GainConstant	Small signal gain constant, in volts per degree	1		Float	NO	(-∞:0) or (0:∞)
MaxAngle	Maximum unwrapped phase angle (+/- MaxAngle) for PhaseCharacteristicType = PhaseFreq	360		Float	NO	

Input Ports

Port	Name	Description	Signal Type	Optional
1	s1	input signal 1	envelope	NO
2	s2	input signal 2	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output signal	envelope	NO

Notes/Equations

1. PhaseCompRF models a phase comparator which compare two complex envelope signals. The output can be a linear difference or a difference that has sine or triangular characteristics.
2. This model reads 1 sample from the inputs and writes 1 sample to the output.
3. This model is useful in the design of RF phase lock loops.
4. This model requires that both inputs are complex envelope signals with characterization frequencies (fc1, fc2) greater than zero. Real input signals are not allowed by this model. If use with a real is needed, then the input to this model can be preceeded with an *EnvFcChange* (algorithm) model that will recharacterize a real signal to its representation at a specified frequency. The simulation rate for this model must be large enough to enable both input signals to be represented at the same characterization frequency as *s1*. Otherwise, a warning message will be displayed. In the following, the phase angles for *s1* and *s2* are discussed after *s2* is converted to its characterization at fc1 (if fc2 is not the same as fc1).
5. The output is calculated as follows:
   θ_1 *(t)* denotes the phase angle associated with *s1*.

   θ_2 *(t)* denotes the phase angle associated with *s2*.

   K is the GainConstant parameter value.
   The output signal at pin 3 is

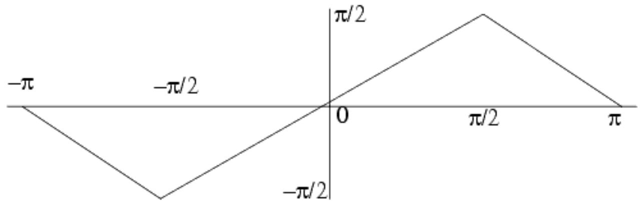
$$output(t) = \begin{cases} K\frac{180}{\pi}\cdot(\theta_1(t)-\theta_2(t)) & \text{for Type = PhaseFreq, MaxAngle = 0} \\ K\frac{180}{\pi}\cdot(diff(t)) & \text{for Type = PhaseFreq, MaxAngle greater than 0} \\ K\frac{180}{\pi}\cdot\sin(\theta_1(t)-\theta_2(t)) & \text{for Type = Sinusoidal} \\ K\frac{180}{\pi}\cdot triangular(\theta_1(t)-\theta_2(t)) & \text{for Type = Triangular} \end{cases}$$

   .

   diff(t) is the difference between the unwrapped phase between s1 and s2 with modulo MaxAngle.

The triangular() function is shown below.

Phase Comparator Characteristics



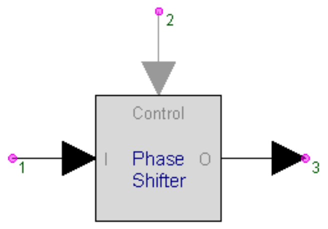
PhaseShifter Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
PhaseShifter (algorithm)	Phase Shifter

PhaseShifter



Description: Phase Shifter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *PhaseShifter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
PhaseShift	Phase shift angle (used when the optional control input is not used)	0	deg	Float	NO	(-∞:∞)
Sensitivity	Phase shift sensitivity in angle/Volt (used when the optional control input is used)	90	deg	Float	NO	(-∞:∞)
HilbertFilterLength	Hilbert filter sample length (used when input is a real signal)	64		Integer	NO	(0:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	envelope	NO
2	control	optional control signal	envelope	YES

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output signal	envelope	NO

Notes/Equations

1. PhaseShifter shifts the phase of the input signal by D degrees. If the control pin is connected, D is the product of *Sensitivity* and the control pin input, otherwise D is taken from *PhaseShift*.
2. This model reads 1 sample from the inputs and writes 1 sample to the output.
3. If the input signal is a complex envelope signal with characterization frequency, fc1, greater than zero, then
   - $v_2(t) = v_1(t)\exp(jD\frac{\pi}{180}); v_1(t) = v_{I_1}(t) + jv_{Q_1}(t)$
4. If the input signal is a real baseband signal, then
   - Define a Hilbert FIR filter with *HilbertFilterLength* number of coefficients.
     Let v1h(t) represent the result of signal v1(t) convolved with this Hilbert filter.
     Let v1d(t) represent the signal v1(t) delayed by *HilbertFilterLength* / 2 number of samples.
   - The signal v2(t) is then obtained as:
     - $v_2(t) = v_{1d}(t) \cdot \cos(D\frac{\pi}{180}) - v_{1h}(t) \cdot \sin(D\frac{\pi}{180})$
   - For good resolution, set *HilbertFilterLength* to N * SR / CenterFrequency.
     where
     SR = simulation sample rate for this model instance.
     CenterFrequency = the center frequency for the information of interest in the real baseband signal.
     N = integer greater than or equal to 1; for better resolution set N > 1.
   - This operation results in a delay of approximately *HilbertFilterLength* / 2 number of samples.

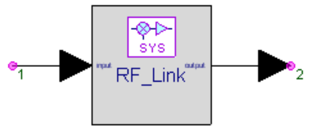
RF_Link Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
RF_Link (algorithm)	Link to Spectrasys RF Design

RF_Link



Description: Link to Spectrasys RF Design
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *RF Link Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Schematic	Spectrasys design to cosimulate with			Text	NO
InputPartName	Input part name (path start)			Text	NO
OutputPartName	Output part name (path end)			Text	NO
FreqSweepSetup	Frequency sweep setup: Automatic, Manual	Automatic		Enumeration	NO
FreqStart	Frequency sweep start	1.0e6	Hz	Float	NO
FreqStop	Frequency sweep stop	2.0e6	Hz	Float	NO
FreqStep	Frequency sweep stop	1.0e4	Hz	Float	NO
FreqSweepPower	Power level for frequency sweep	1.0e-3	W	Float	NO
EnableNoise	Enable thermal noise extraction: NO, YES	NO		Enumeration	NO
Temperature	System temperature	16.85	degC	Float	NO

Input Ports

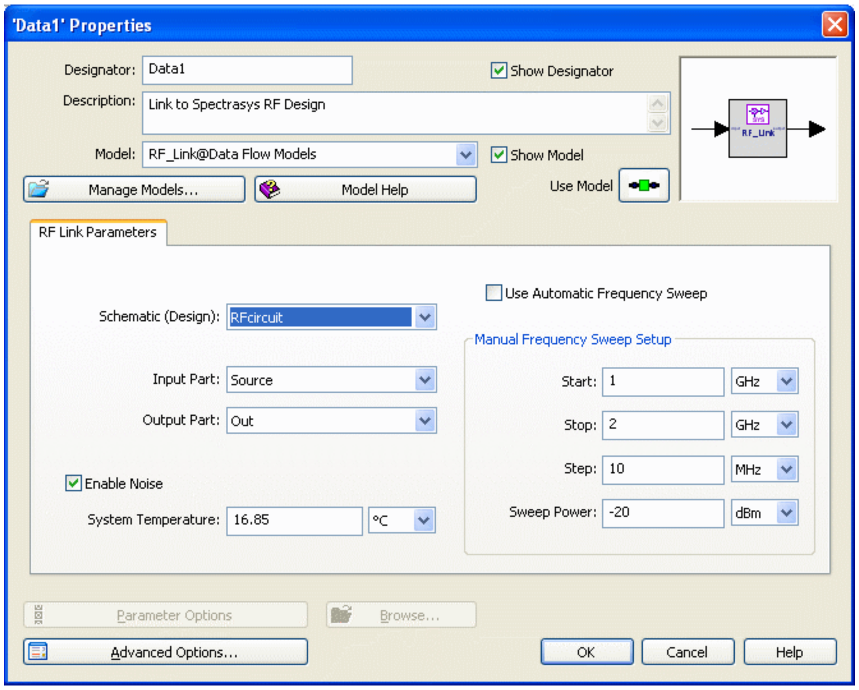
Port	Name	Description	Signal Type	Optional
1	input	input signal	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	envelope	NO

Setup UI

A custom UI is provided for this model that allows easy set up.


'Data1' Properties

Designator: Data1 [✓] Show Designator

Description: Link to Spectrasys RF Design

Model: RF_Link@Data Flow Models [✓] Show Model

RF Link Parameters

Schematic (Design): RFcircuit

Input Part: Source

Output Part: Out

[✓] Enable Noise

System Temperature: 16.85 °C

[ ] Use Automatic Frequency Sweep

Manual Frequency Sweep Setup

Start: 1 GHz

Stop: 2 GHz

Step: 10 MHz

Sweep Power: -20 dBm

- The **Schematic** drop down box lists all the Spectrasys designs in the workspace. As soon as a schematic is selected, the **Input Part** and **Output Part** drop down boxes are updated to list all the available input and output part names that can be used to define the path of interest.
- The **Enable Noise** checkbox can be used to enable/disable thermal noise analysis for the Spectrasys design.
- The **Use Automatic Frequency Sweep** check box can be used to enable/disable the automatic frequency sweep setup.
  - If checked, then the frequency sweep used to extract the frequency response for the defined path in the Spectrasys design is setup automatically based on the input signal sampling rate and characterization frequency.
  - If not checked, then frequency sweep **Start**, **Stop**, and **Step** values as well as the power level at which the frequency sweep is performed (**Sweep Power**) need to be specified in the fields inside the **Manual Frequency Sweep Setup** area.

Notes/Equations

1. The RF_Link model allows simulating a Spectrasys RF design in a Data Flow schematic.
2. During simulation initialization a Spectrasys simulation is run that extracts behavioral data for the Spectrasys RF design. The extracted behavioral data characterizes the Spectrasys RF design in terms of its frequency response, non-linear behavior, and thermal noise performance. This data is used to set up an equivalent Data Flow network consisting of filters, amplifiers, mixers, oscillators, and noise generators.
3. For more details see *About using Spectrasys designs in Data Flow schematics* (sim)

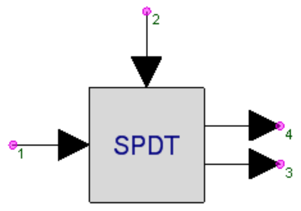
SwitchSPDT Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SwitchSPDT (algorithm)	Single Pole Double Throw Switch

SwitchSPDT



Description: Single Pole Double Throw Switch
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *SwitchSPDT Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss1	Loss in dB for on state insertion loss for output 1	0		Float	NO	[0:+200)
Isolation1	Isolation in dB for off state insertion loss for output 1	200		Float	NO	[0:+200)
Loss2	Loss in dB for on state insertion loss for output 2	0		Float	NO	[0:+200)
Isolation2	Isolation in dB for off state insertion loss for output 2	200		Float	NO	[0:+200)
VThreshold	Control voltage threshold	0.5	V	Float	NO	(0:∞)
TOn1	On state transition time for output 1	0	s	Float	NO	[0:∞)
TOff1	Off state transition time for output 1	0	s	Float	NO	[0:∞)
TOn2	On state transition time for output 2	0	s	Float	NO	[0:∞)
TOff2	Off state transition time for output 2	0	s	Float	NO	[0:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input1	input signal 1	envelope	NO
2	input2	input signal 2	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	output1	output signal 1	envelope	NO
4	output2	output signal 2	envelope	NO

Notes/Equations

1. SwitchSPDT models a single pole double throw switch with non-idealities including insertion loss, imperfect isolation, and non-zero switching time.
2. This model reads 1 sample from the inputs and writes 1 sample to the outputs.
3. The outputs take on the attributes of *input1*. If *input1* is a real baseband signal, the the outputs will be. If the *input1* is a complex envelope signal, then the output will also be with the same characterization frequency. The *input2* is always interpreted as a real baseband signal. If *input2* is a complex envelope signal, it will be converted internally in this model to its equivalent form as a real baseband signal.
4. SwitchSPDT inputs and output are defined as follows.
   Let $v_i(t)$ be the real or complex envelop value at pin i.

   The control input $v_2(t)$ is always forced to a real value for comparison with *VThreshold*.
   Case 1: $v_2(t)$ > *VThreshold* connects pin 1 to pin 4.

   Let T_s be the time at which $v_2(t)$ exceeds *VThreshold*. Then

$$v_3(t) = \begin{cases} 10^{-\frac{ISO_2}{20}} v_1(t) & \text{if } t \geq T_s + T_{off_2} \\ \left(10^{-\frac{LOSS_2}{20}} - 10^{-\frac{ISO_2}{20}}\right)\left(1 - \frac{t-T_s}{T_{off_2}}\right) v_1(t) + 10^{-\frac{ISO_2}{20}} v_1(t) & \text{otherwise} \end{cases}$$

$$v_4(t) = \begin{cases} 10^{-\frac{LOSS_1}{20}} v_1(t) & \text{if } t \geq T_s + T_{on_1} \\ \left(10^{-\frac{LOSS_1}{20}} - 10^{-\frac{ISO_1}{20}}\right) \frac{t-T_s}{T_{on_1}} v_1(t) + 10^{-\frac{ISO_1}{20}} v_1(t) & \text{otherwise} \end{cases}$$

Case 2: *v* $_2$*(t)* ≤ *VThreshold* connects pin 1 to pin 3.

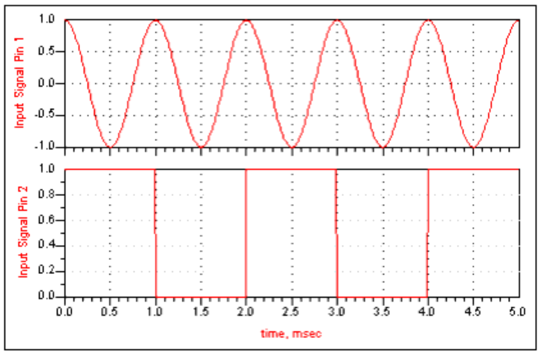
Let *T* $_s$ be the time at which *v* $_2$*(t)* falls below *VThreshold*. Then

$$v_3(t) = \begin{cases} 10^{-\frac{LOSS_2}{20}} v_1(t) & \text{if } t \geq T_s + T_{on_1} \\ \left(10^{-\frac{LOSS_2}{20}} - 10^{-\frac{ISO_2}{20}}\right) \frac{t-T_s}{T_{on_2}} v_1(t) + 10^{-\frac{ISO_2}{20}} v_1(t) & \text{otherwise} \end{cases}$$

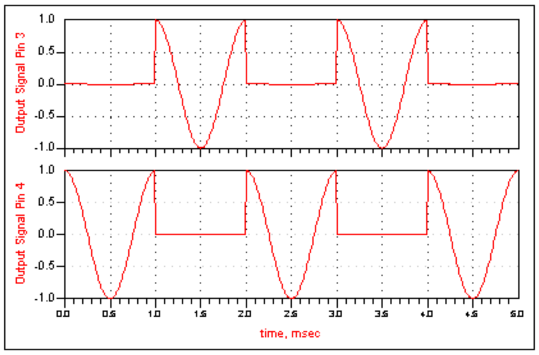
$$v_4(t) = \begin{cases} 10^{-\frac{ISO_1}{20}} v_1(t) & \text{if } t \geq T_s + T_{off_1} \\ \left(10^{-\frac{LOSS_1}{20}} - 10^{-\frac{ISO_1}{20}}\right)\left(1 - \frac{t-T_s}{T_{off_1}}\right) v_1(t) + 10^{-\frac{ISO_1}{20}} v_1(t) & \text{otherwise} \end{cases}$$

A SwitchSPDT Example

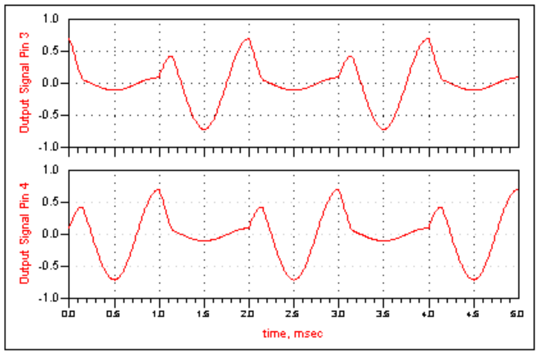
1. The SwitchSPDT example uses a 1 KHz cosine input at pin 1 and a 0.5 KHz square wave control at pin 2.



2. An ideal SwitchSPDT part with the following parameters give the following output.
Loss1=0 dB, Isolation1=200 dB,
Loss2=0 dB, Isolation2=200 dB,
VThreshold=0.50,
TOn1=0 s, TOff1=0 s,
TOn2=0 s, TOff2=0 s



3. A non-ideal SwitchSPDT part with the following parameters give the following output.
Loss1=3 dB, Isolation1=20 dB,
Loss2=3 dB, Isolation2=20 dB,
VThreshold=0.50,
TOn1=0.15 s, TOff1=0.15 s,
TOn2=0.15 s, TOff2=0.15 s


Output Signal Pin 3
Output Signal Pin 4
1.0
0.5
0.0
-0.5
-1.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
time, msec

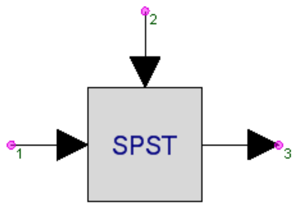
SwitchSPST Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SwitchSPST (algorithm)	Single Pole Single Throw Switch

SwitchSPST



Description: Single Pole Single Throw Switch
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *SwitchSPST Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Loss in dB for on state insertion loss	0		Float	NO	[0:+200)
Isolation	Isolation in dB for off state insertion loss	-200		Float	NO	[0:+200)
VThreshold	Control voltage threshold	0.5	V	Float	NO	(0:∞)
TOn	On-state transition ctime for output	0	s	Float	NO	[0:∞)
TOff	Off-state transition ctime for output	0	s	Float	NO	[0:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input1	input signal 1	envelope	NO
2	input2	input signal 2	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output signal	envelope	NO

Notes/Equations

1. SwitchSPST models a single pole single throw switch with non-idealities including insertion loss, imperfect isolation, and non-zero switching time.
2. This model reads 1 sample from the inputs and writes 1 sample to the output.
3. The output take on the attributes of *input1*. If *input1* is a real baseband signal, the the output will be. If the *input1* is a complex envelope signal, then the output will also be with the same characterization frequency. The *input2* is always interpreted as a real baseband signal. If *input2* is a complex envelope signal, it will be converted internally in this model to its equivalent form as a real baseband signal.
4. SwitchSPST inputs and output are defined as follows
   Let v i(t) be the real or complex envelop value at pin i.
   The control input v 2(t) is always forced to a real value for comparison with *VThreshold*.
   Case 1: $v_2(t) > VThreshold$

   Let T_s be the time at which $v_2(t)$ exceeds *VThreshold*.

   Then

$$v_3(t) = \begin{cases} 10^{-\frac{Loss}{20}} v_1(t) & if\, t \ge T_s + Ton \\ \left(10^{-\frac{Loss}{20}} - 10^{-\frac{Iso}{20}}\right)\frac{(t-T_s)}{Ton} v_1(t) + 10^{-\frac{Iso}{20}} v1(t) & otherwise \end{cases}$$

   Case 2: $v_2(t) \le VThreshold$

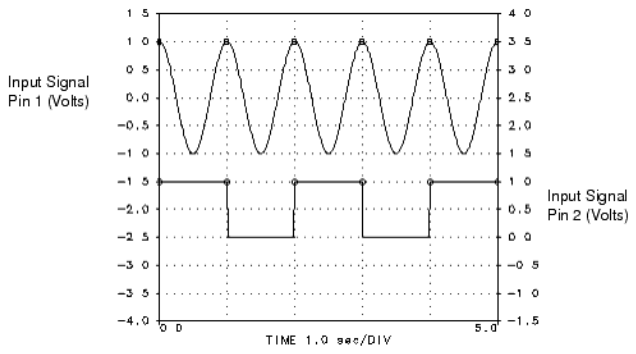
   Let T_s be the time at which $v_2(t)$ falls below *VThreshold*.

Then

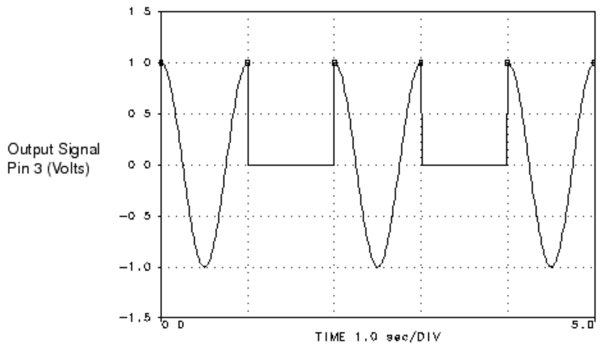
$$v_3(t) = \begin{cases} 10^{-\frac{ISO}{20}} v_1(t) & if\ t \ge T_s + Toff \\ \left(10^{-\frac{LOSS}{20}} - 10^{-\frac{ISO}{20}}\right)\left(1 - \frac{(t - T_s)}{Toff}\right) v_1(t) + 10^{-\frac{ISO}{20}} v_1(t) & otherwise \end{cases}$$

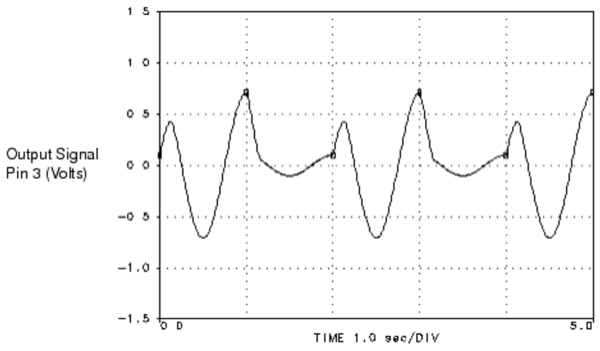
5. SwitchSPST Component Inputs and SwitchSPST Component Output show the performance of the SwitchSPST component, when it is an ideal switch, using the following parameter values:
Loss=0, Iso=200, Vth=0.50, Ton=0, Toff=0
SwitchSPST Component Output shows the output of the SwitchSPST component, when the switch is not ideal, using the parameter values:
Loss=3 dB, Iso=20 dB, Vth=0.50, Ton=0.15 s, Toff=0.15 s

SwitchSPST Component Inputs

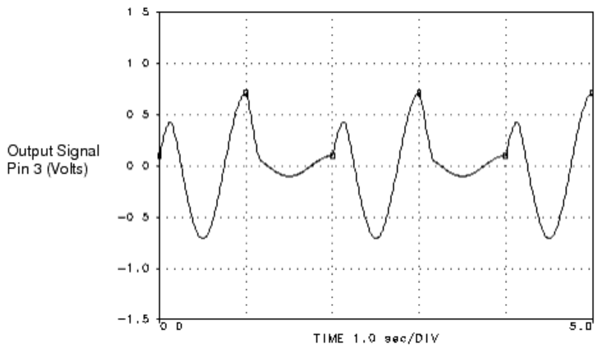


SwitchSPST Component Output





SwitchSPST Component Output


Output Signal
Pin 3 (Volts)
1.5
1.0
0.5
0.0
-0.5
-1.0
-1.5
0.0
5.0
TIME 1.0 sec/DIV

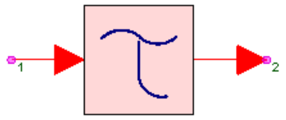
TimeDelay Part

Categories: *Analog/RF* (algorithm), *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
TimeDelay (algorithm)	Ideal Time Delay Block. Delays the signal for a certain amount of time.

TimeDelay



Description: Ideal Time Delay Block. Delays the signal for a certain amount of time.
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *TimeDelay Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Unit	Time delay unit: Time, TimeStep	Time		Enumeration	NO	
T	Delay in time	0	s	Float	NO	[0, ∞)
N	Delay in number of time steps	0		Integer	NO	[0, ∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Notes/Equations

1. TimeDelay delays the signal in time.
2. For every input, one value is output.
3. If Unit is *Time*, the input is delayed by T where T is any non-negative number.
4. Otherwise, the input is delayed by N × time step where N is any non-negative integer.
5. TimeDelay increases the time stamps by the delay without inserting zero valued samples.
6. See page on *Special Models in Timing Method* (sim).

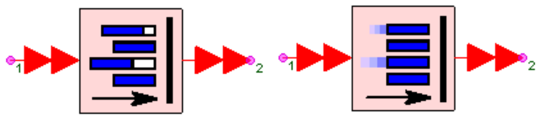
TimeSynchronizer Part

Categories: *Analog/RF* (algorithm), *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
TimeSynchronizer (algorithm)	Synchronize signals in time

TimeSynchronizer



Description: Synchronize signals in time
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *TimeSynchronizer Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Mode	Time synchronization mode: ZeroPadding, TimeDelay	ZeroPadding		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	multiple anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	multiple anytype	NO

Notes/Equations

1. TimeSynchronizer aligns multiple inputs in time.
2. This model reads 1 sample from each of the inputs and writes 1 sample to each of the outputs.
3. The _i_th input flows to _i_th output, therefore the number of the input connections must at least equal to the number of the output connections.
4. If Mode is *ZeroPadding*, zero valued samples are inserted to later arriving inputs such that all signals are aligned in time to the earliest incoming signal.
5. Otherwise, the earlier incoming signal time stamps are adjusted without inserting zeros such that all signals are aligned to the latest incoming signal.
6. See section on *Special Models in Timing Method* (sim).

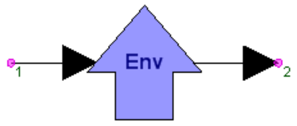
UpSampleEnv Part

Categories: *Analog/RF* (algorithm), *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
UpSampleEnv (algorithm)	Up Sampler for Envelope Signal

UpSampleEnv



Description: Up Sampler for Envelope Signal
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *UpSampleEnv Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Factor	Upsampling ratio	5		Integer	NO	[1:∞)
Mode	Upsampling interpolation type: Insert zeros, Hold sample, Polyphase filter, Linear	Hold sample		Enumeration	NO	
Phase	Upsampling insertion phase for the output non-zero sample	0		Integer	NO	[0:Factor - 1]
ExcessBW	Excess bandwidth of raised cosine interpolation filter	0.5		Float	NO	[0:1]

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	envelope	NO

Output Ports

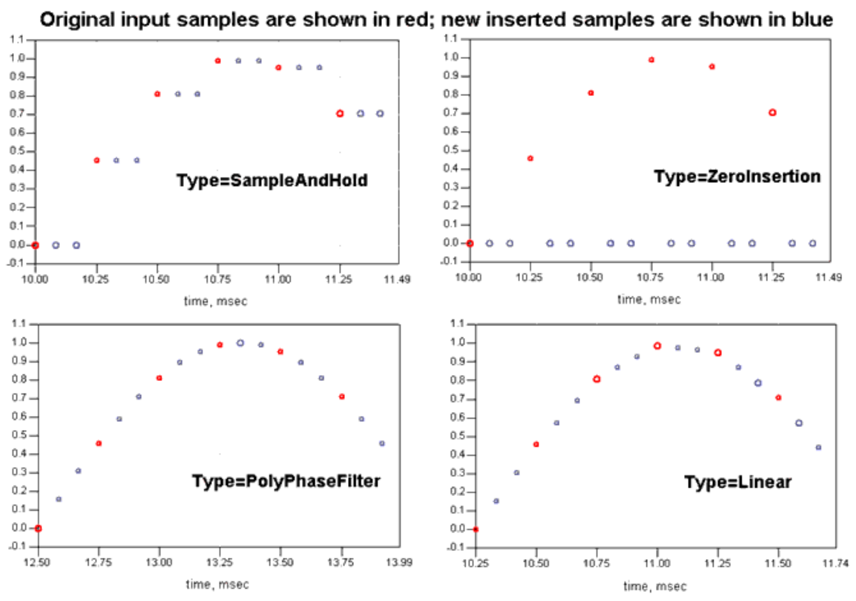
Port	Name	Description	Signal Type	Optional
2	output	output signal	envelope	NO

⚠ To avoid confusion on a schematic, it is best to **mirror** (flip) this symbol (instead of **rotating** it) when you need to have the input on the *right* and the output on the *left* of the symbol; otherwise the symbol arrow will point in the "wrong" direction. Select the part and press F6 to flip the symbol into the correct orientation.

Notes/Equations

1. This model upsamples an input with time step *TStepIn* to the output that has time step *TStepOut* = *TStepIn* / *Factor*.
2. This model reads 1 sample from the input and writes *Factor* number of samples to the output.
3. If Mode is *Insert zeros*, the Phase parameter identifies which of the Factor output samples will contain the one input sample.
4. If Mode is *Hold samples*, the input sample is repeated Factor times at the output.
5. If Mode is *Polyphase filter*, a raised-cosine filter with excess bandwidth equal to ExcessBW (ExcessBW = 0 gives the ideal lowpass filter) is used for interpolation. The corner frequency of the filter is set to FCorner = 1 / Factor / TStepIn. The number of taps used for this filter is (1 + 20 × Factor). Therefore, the output signal will be delayed with respect to the input signal by 10 × TStepIn.
6. If Mode is *Linear*, linear interpolation is used to fill the values between consecutive input samples. The output signal will be delayed by TStepIn with respect to the input signal.
7. To upsample a signal by a non-integer factor, a cascade of an upsampler followed by a downsampler is needed. For example, to change the sampling rate of a signal from 40 MHz (TStep = 25 nsec) to 73 MHz (TStep = 13.69863 nsec), first pass it through the UpSampleEnv component (Factor = 73, Mode = *Polyphase filter* or *Linear*) and then through the DownSampleEnv component (Factor = 40). To improve simulation speed, make sure the two ratios are relatively prime. If not, divide them with their GCD (greatest common divisor). For example, if in the previous case the desired sampling rate was 72 MHz, Factor of UpSampleEnv can be set to 9 (= 72 / 8) and Factor of DownSampleEnv can be set to 5 (= 40 / 8) where GCD (72, 40) = 8.
8. The figure below shows the UpSampleEnv output for different Mode values.

Upsampler outputs for the different Mode values: *Insert Zeros*(Type=ZeroInsertion), *Hold samples* (Type=SampleAndHold), *Polyphase filter*(Type=PolyPhaseFilter) and *Linear*(Type=Linear). The ZeroInsertion plot was created with *Phase* set to 0. The PolyPhaseFilter plot wascreated with ExcessBW set to 0.5.



Note on large upsampling factors

Each upsampler require a buffer of Factor number of samples. For a large Factor value, memory requirements may prevent a simulation. The way around this problem is to substitute a cascade of upsamplers. For example, a Factor of 10^6 would require a buffer of 10^6 samples. If a cascade of two upsamplers were used, then each Factor could be 10^3 which would require a total buffer equivalent of 2×10^3 samples. If a cascade of three upsamplers were used, then each Factor could be 10^2 which would require a total buffer equivalent of 3×10^2 samples.
If a nonzero Phase were to be used, the Phase would have to be deconstructed into Phase values for each cascaded upsampler. Let an upsampler have a Factor of 10^6 and a Phase of 123456. The upsampler is substituted by 3 upsamplers each having a Factor of 10^2. The first upsampler would have a Phase of 12, the second upsampler would have a Phase of 34 and the third sampler would have a Phase of 56.

See:
DownSampleEnv (algorithm)

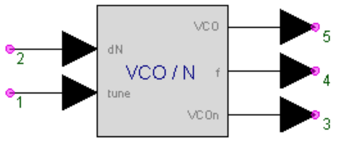
VCO_DivideByN Part

Categories: *Analog/RF* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
VCO_DivideByN (algorithm)	VCO with Internal Divide by N Divider

VCO_DivideByN



Description: VCO with Internal Divide by N Divider
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *VCO DivideByN Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
FCarrier	VCO carrier frequency	1000000	Hz	Float	NO	(0:∞)
Power	Unmodulated carrier power	0.01	W	Float	NO	[0:∞)
Sensitivity	Frequency deviation sensitivity in Hz/volts	1		Float	NO	(-∞:∞)
N	Nominal divide-by-N ratio	1		Float	NO	(0:∞)
RefR	Reference resistance	50	ohm	Float	NO	(0:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	vtune	input tune signal	envelope	NO
2	dN	input delta divide by N ratio	envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	VCOn	output VCO divide by N+dN RF signal	envelope	NO
4	Freq	output VCO instantaneous frequency	envelope	NO
5	VCO	output VCO undivided RF signal	envelope	NO

Notes/Equations

1. VCO_DivideByN models a VCO with a built-in controllable frequency divider. It outputs the VCO output signal (*VCO*) as well as the frequency divide signal (*VCOn*).
2. This model reads 1 sample from the inputs and writes 1 sample to the outputs.
3. Incorporating the divider into the same model permits its use in phase-lock loop simulations where the the primary interest is in the *VCOn* signal which requires a smaller sampling rate than does the *VCO* signal. This model does not check for sample rate validity for the *VCO* signal. If the sample rate is too small, the phase and frequency information of the *VCO* output may be aliased. If just the divided output *VCOn* is used, the loop simulations can still be valid and simulate faster due to the smaller sample rate required.
   As the sample rate is increased such that the *VCO* signal is not aliased, then both the *VCO* and divided *VCOn* outputs are valid.
4. The inputs *vtune* and *dN* are interpreted as real baseband signals.
5. The outputs *VCO* and *VCOn* are complex envelope signals with characterization frequencies *FCarrier* and *FCarrier* / *N* respectively.
6. The output *Freq* is a real baseband signal whose value is the frequency of the *VCO* as determined by *FCarrier* + *Sensitivity* * *vtune*.
7. The divide number is determined by adding the *N* parameter and the *dN* baseband input voltage. The divide number can change during the simulation. By properly driving the *dN* input, fractional frequency division can be simulated. To simulate all the dynamics of a fractional divider, the simulation sample rate must be large enough to properly digitize the varying divide rate.
8. Both the main *VCO* output and the divided output *VCOn* have unmodulated carrier voltage levels determined from the *Power* and *RefR* parameter values.

C++ Code Generation

The parts listed in this category have a default model that supports C++ Code Generation. If the part has more than one model, the non-default models do not necessarily support C++ Code Generation. To find out whether a specific model supports C++ Code Generation go to its documentation page and look for the *C++ Code Generation Support* field.

For more information about C++ Code generation, refer to *C++ Code Generation* (algorithm).

Contents

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 shells (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.

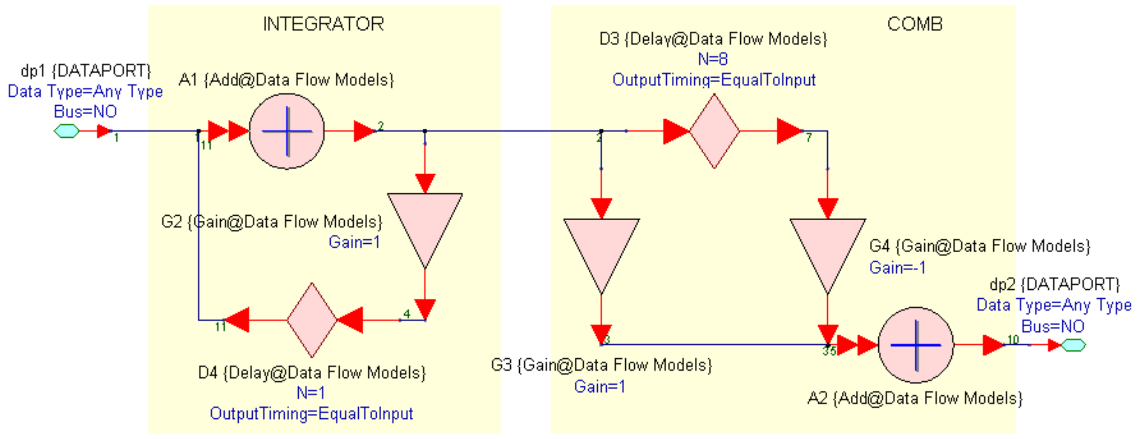
The generated C++ Model is also generic enough to be used as standalone C++ code for other applications.

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

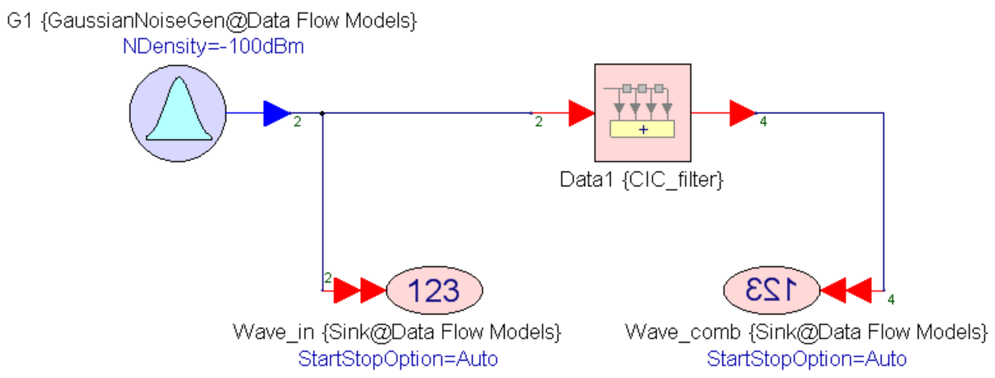
Create a sub-network model implementing a CIC filter using models in Algorithm Design library as follows,



INTEGRATOR
COMB
dp1 {DATAPORT}
Data Type=Any Type
Bus=NO
A1 {Add@Data Flow Models}
D3 {Delay@Data Flow Models}
N=8
OutputTiming=EqualToInput
G2 {Gain@Data Flow Models}
Gain=1
G4 {Gain@Data Flow Models}
Gain=-1
dp2 {DATAPORT}
Data Type=Any Type
Bus=NO
D4 {Delay@Data Flow Models}
N=1
OutputTiming=EqualToInput
G3 {Gain@Data Flow Models}
Gain=1
A2 {Add@Data Flow Models}

Creating a Design using the Sub-network Model

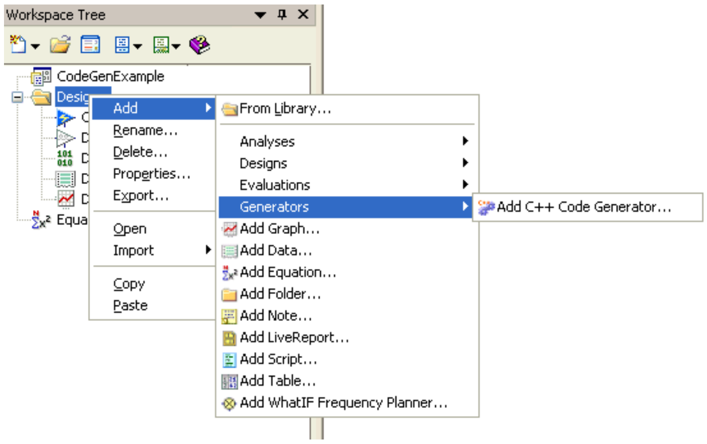
Create a Design using the CIC filter sub-network model as follows



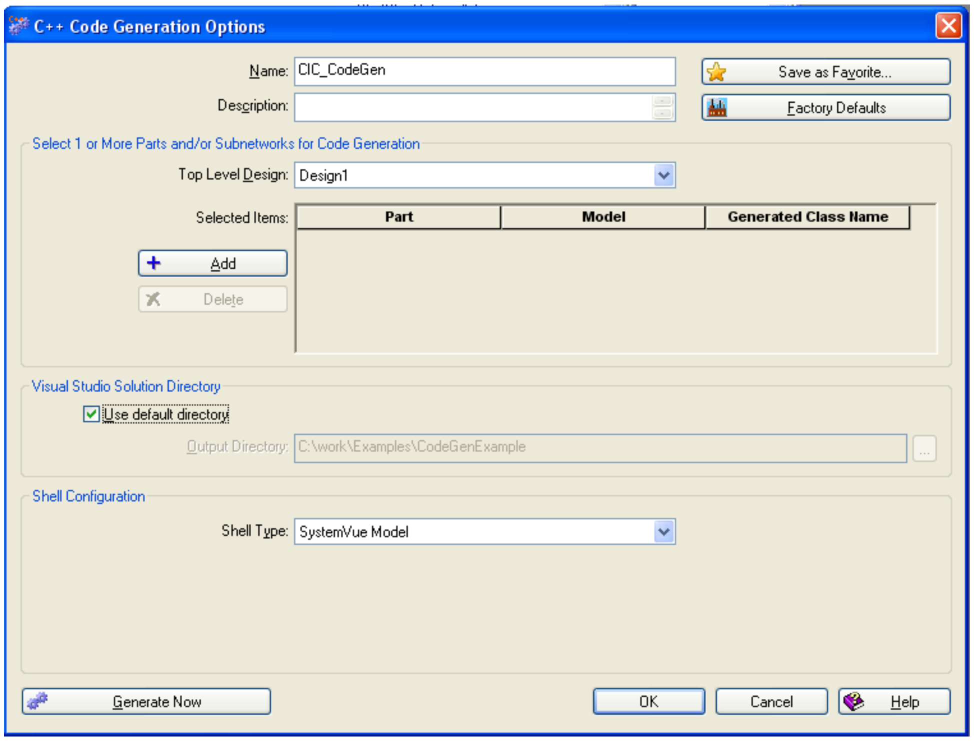
G1 {GaussianNoiseGen@Data Flow Models}
NDensity=-100dBm
Data1 {CIC_filter}
123
Wave_in {Sink@Data Flow Models}
StartStopOption=Auto
Wave_comb {Sink@Data Flow Models}
StartStopOption=Auto

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_CodeGen and select Top Level Design to be Design1. The dialog should look as shown below:


C++ Code Generation Options
Save as Favorite...
Factory Defaults
Select 1 or More Parts and/or Subnetworks for Code Generation
Add
Delete
Visual Studio Solution Directory
Shell Configuration
Generate Now
OK
Cancel
Help

Name: CIC_CodeGen
Description:
Top Level Design: Design1
Selected Items:

Part	Model	Generated Class Name

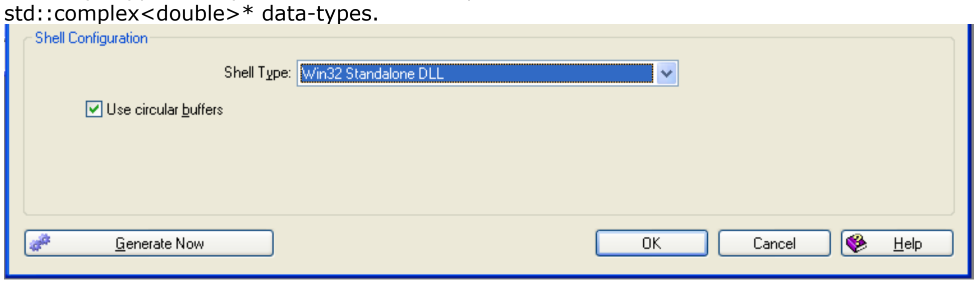
[✓] Use default directory
Output Directory: C:\work\Examples\CodeGenExample
Shell Type: SystemVue Model

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 **Top Level Design** selects the top level design for code generation.
- 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 **Shell Type** drop down menu specifies the shell (application) on which the

generated code will run. The available shells are: SystemVue Model, Win32 Standalone DLL, ADS Ptolemy Model. See Supported Shells 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 shell 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.

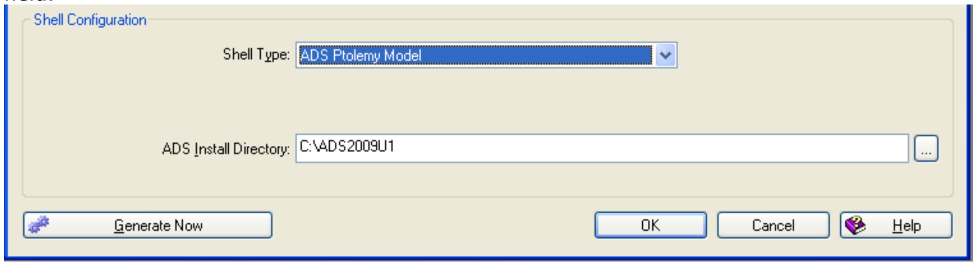


Shell Configuration

Shell Type: Win32 Standalone DLL

[✓] Use circular buffers

- **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 shell the **ADS Install Directory** needs to be specified in the corresponding field.

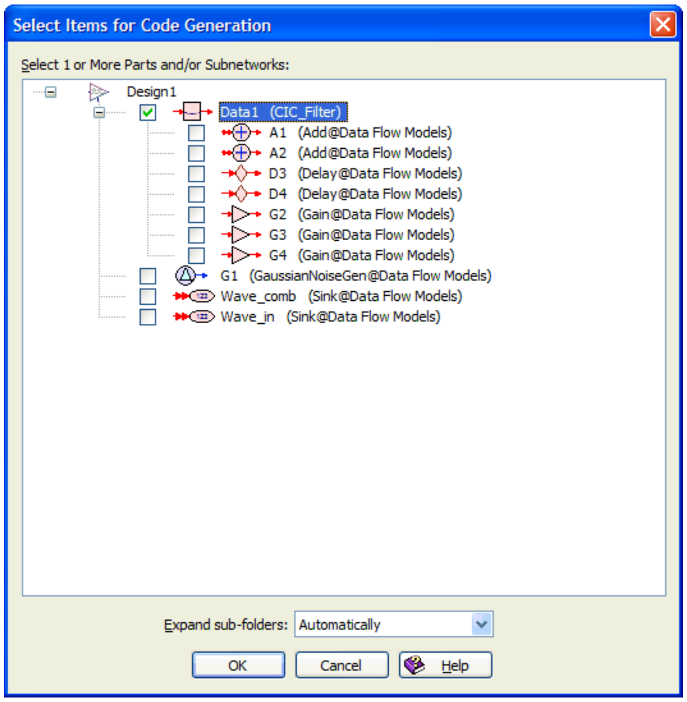


Shell Configuration

Shell Type: ADS Ptolemy Model

ADS Install Directory: C:\ADS2009U1

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



Select Items for Code Generation

Select 1 or More Parts and/or Subnetworks:

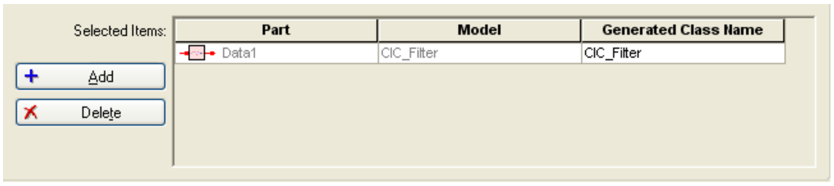
Design1
- [✓] Data1 (CIC_Filter)
  - [ ] A1 (Add@Data Flow Models)
  - [ ] A2 (Add@Data Flow Models)
  - [ ] D3 (Delay@Data Flow Models)
  - [ ] D4 (Delay@Data Flow Models)
  - [ ] G2 (Gain@Data Flow Models)
  - [ ] G3 (Gain@Data Flow Models)
  - [ ] G4 (Gain@Data Flow Models)
- [ ] G1 (GaussianNoiseGen@Data Flow Models)
- [ ] Wave_comb (Sink@Data Flow Models)
- [ ] Wave_in (Sink@Data Flow Models)

Expand sub-folders: Automatically

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.



Part	Model	Generated Class Name
Data1	CIC_Filter	CIC_Filter

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** shell) 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 shells (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.

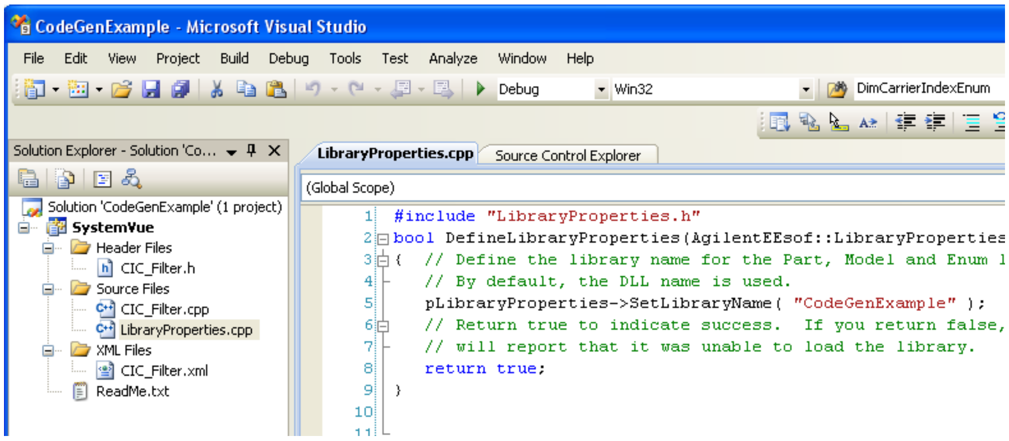
When selecting a C__ model part (non-subnetwork part) for code generation, SystemVue will generate a C__ wrapper inherited from the original C__ model. For this reason, the **Generated Class Name** should be different than the full class name (including namespace) of the original C__ model. In general,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

Supported Shells

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

SystemVue Model

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



```
#include "LibraryProperties.h"
bool DefineLibraryProperties(AgilentEEsof::LibraryProperties
{   // Define the library name for the Part, Model and Enum l
    // By default, the DLL name is used.
    pLibraryProperties->SetLibraryName( "CodeGenExample" );
    // Return true to indicate success.  If you return false,
    // will report that it was unable to load the library.
    return true;
}
```

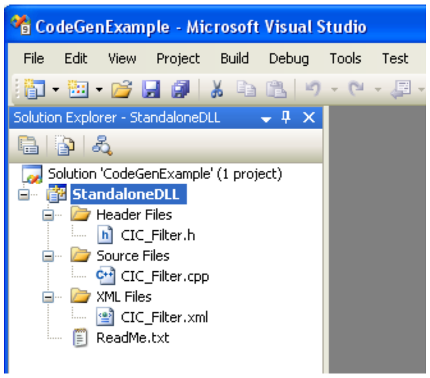
The solution name is the same as the workspace name (CodeGenExample) and the project name is SystemVue.

- 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 shell 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.


CodeGenExample - Microsoft Visual Studio
File Edit View Project Build Debug Tools Test
Solution Explorer - StandaloneDLL
Solution 'CodeGenExample' (1 project)
StandaloneDLL
Header Files
CIC_Filter.h
Source Files
CIC_Filter.cpp
XML Files
CIC_Filter.xml
ReadMe.txt

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

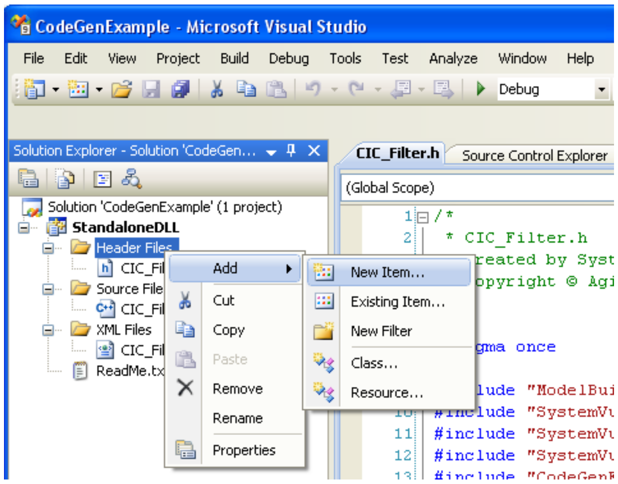
- The *Header Files* folder contains the header file(s) for the generated classes.
- The *Source Files* folder contains the implementation (.cpp) file(s) for the generated classes.
- The *XML Files* folder contains xml file(s) that describe the model interface, that is, the names, types, and other properties 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:

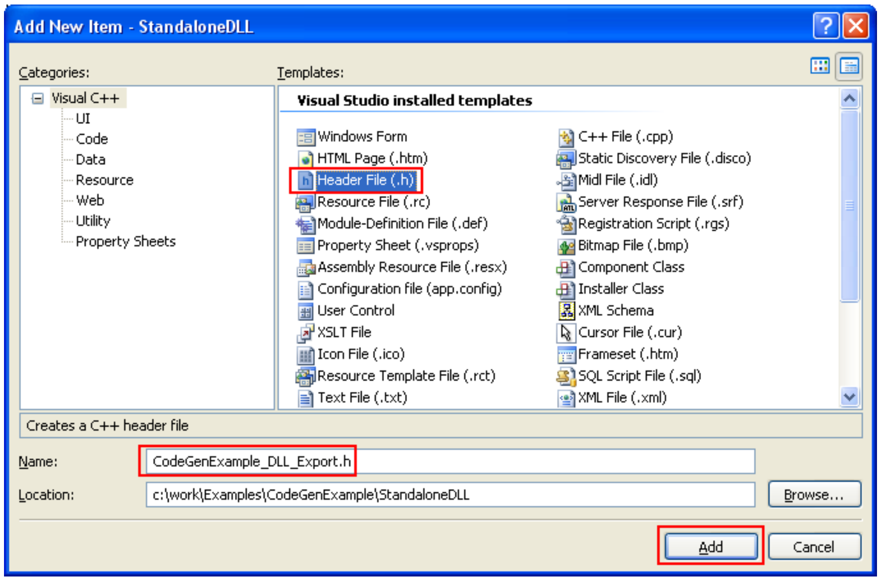
```
__declspec(dllexport)
```

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 ...**



```
/*
 * CIC_Filter.h
```

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.



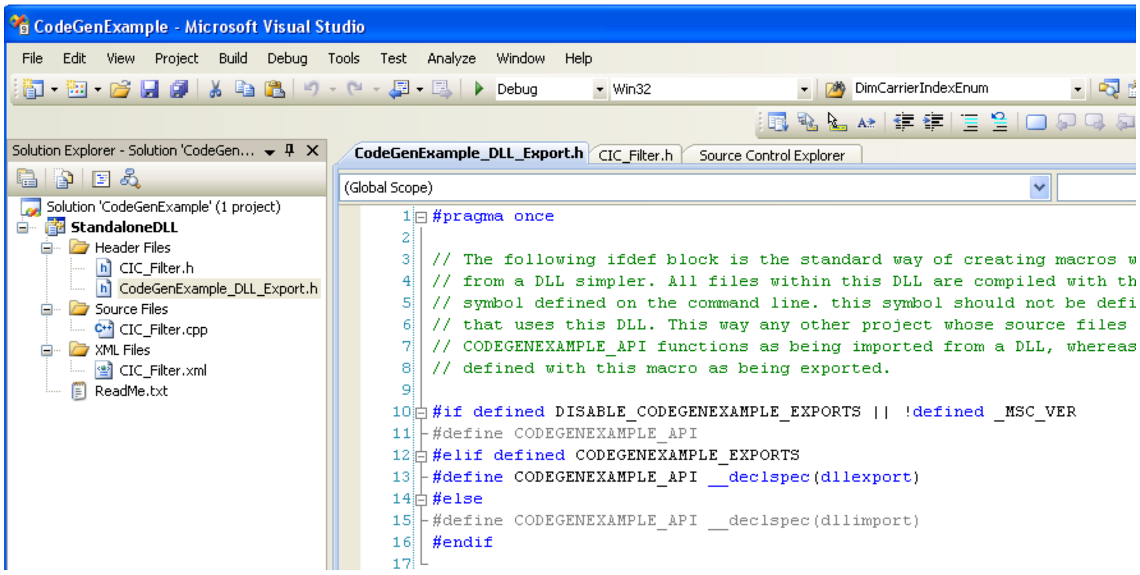
Creates a C++ header file

Name: CodeGenExample_DLL_Export.h

Location: c:\work\Examples\CodeGenExample\StandaloneDLL

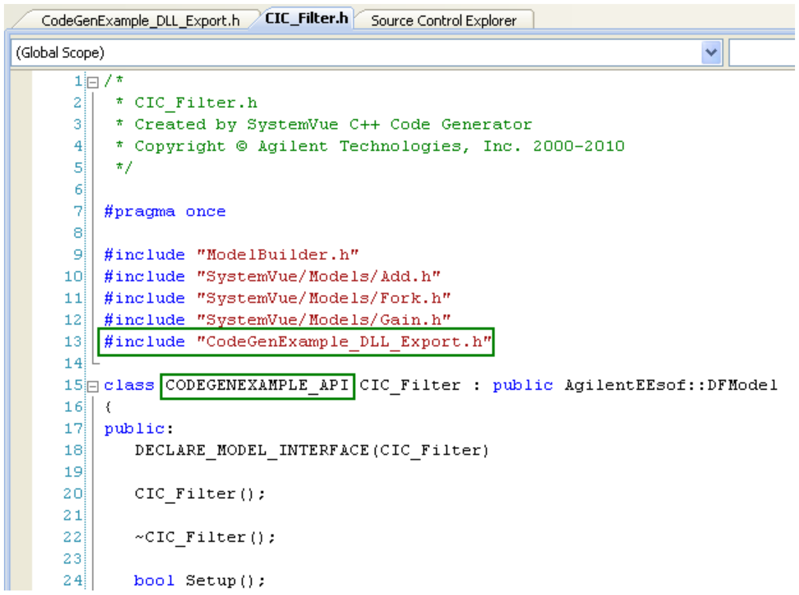
An empty file called CodeGenExample_DLL_Export.h is created and opened in Visual Studio for editing. Copy the content shown below and paste it in this file. Replace CODEGENEXAMPLE with the name of your solution.

```
#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
```



```
#pragma once

// The following ifdef block is the standard way of creating macros w
// from a DLL simpler. All files within this DLL are compiled with th
// symbol defined on the command line. this symbol should not be defi
// that uses this DLL. This way any other project whose source files
// CODEGENEXAMPLE_API functions as being imported from a DLL, whereas
// 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.



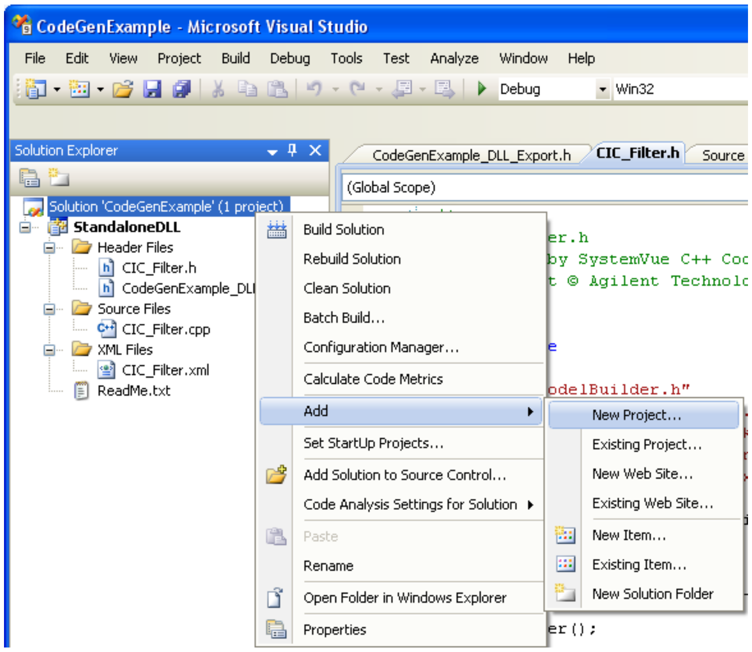
```
/*
 * CIC_Filter.h
 * Created by SystemVue C++ Code Generator
 * Copyright © Agilent Technologies, Inc. 2000-2010
 */

#pragma once

#include "ModelBuilder.h"
#include "SystemVue/Models/Add.h"
#include "SystemVue/Models/Fork.h"
#include "SystemVue/Models/Gain.h"
#include "CodeGenExample_DLL_Export.h"

class CODEGENEXAMPLE_API CIC_Filter : public AgilentEEsof::DFModel
{
public:
   DECLARE_MODEL_INTERFACE(CIC_Filter)

   CIC_Filter();

   ~CIC_Filter();

   bool Setup();
```

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.

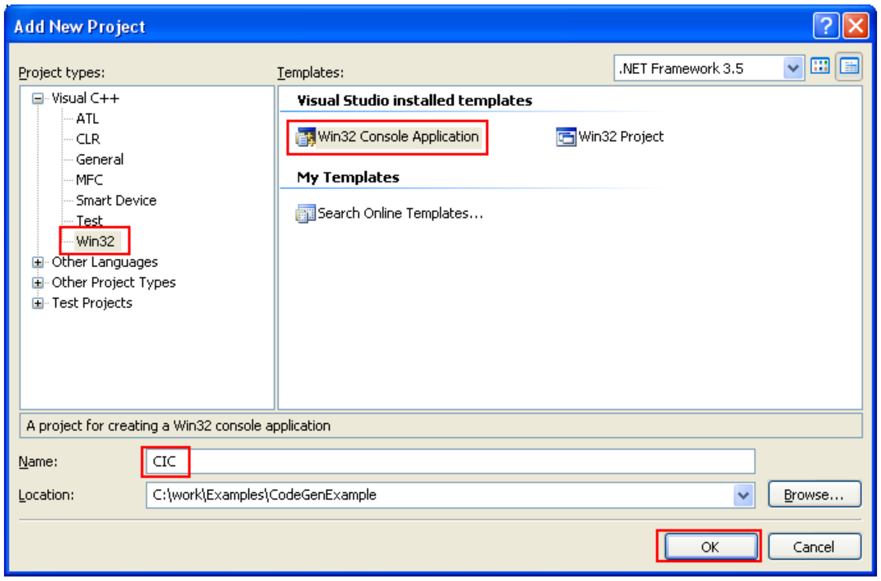
Now the CIC_Filter class/model can be instantiated and used in other applications. An example on how to use a model from a standalone DLL in a standalone executable is described in the next section.

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 ...**



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.

3. 

Project types:
- Visual C++
  - ATL
  - CLR
  - General
  - MFC
  - Smart Device
  - Test
  - Win32
- Other Languages
- Other Project Types
- Test Projects

Templates: .NET Framework 3.5

Visual Studio installed templates: Win32 Console Application, Win32 Project

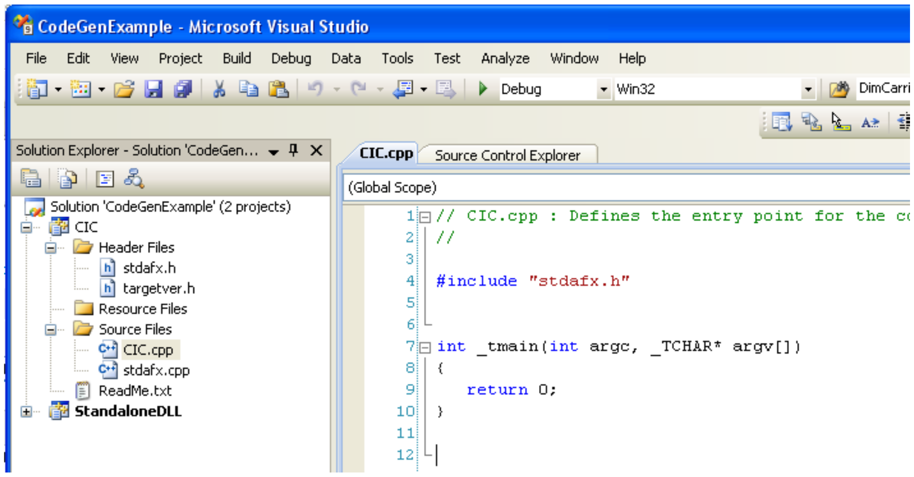
My Templates: Search Online Templates...

A project for creating a Win32 console application

Name: CIC

Location: C:\work\Examples\CodeGenExample

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.

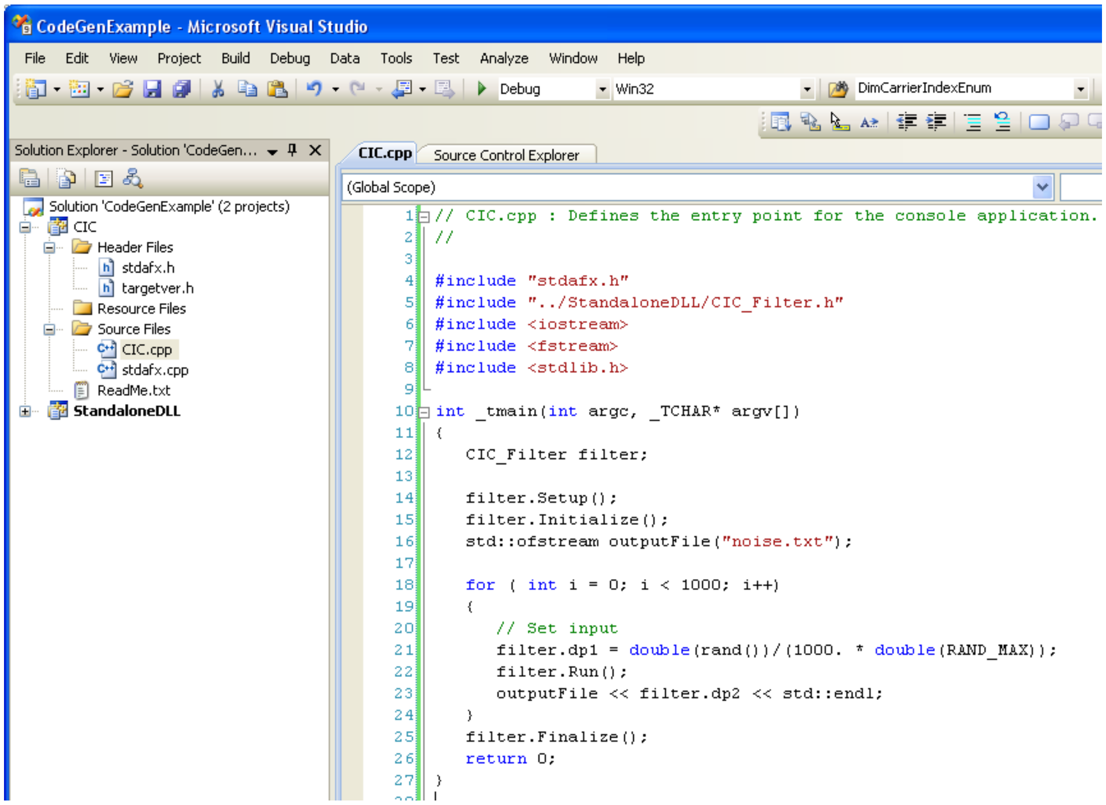


CodeGenExample - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Win32
DimCarri
Solution Explorer - Solution 'CodeGen...
CIC.cpp
Source Control Explorer
(Global Scope)

Solution 'CodeGenExample' (2 projects)
- CIC
  - Header Files
    - stdafx.h
    - targetver.h
  - Resource Files
  - Source Files
    - CIC.cpp
    - stdafx.cpp
  - ReadMe.txt
- StandaloneDLL

```
// CIC.cpp : Defines the entry point for the co
//

#include "stdafx.h"


int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}

```

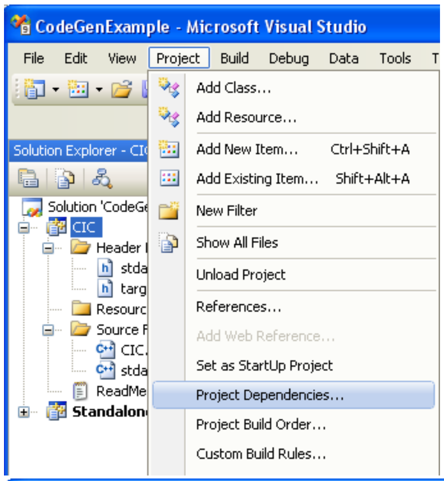
6. Edit this file to implement the application you want. The code shown below (this code is also provide as a code snippet below so that you can copy/paste it and try it out yourself)
   - instantiates the CIC_Filter class (line 12) that was generated by SystemVue
   - initializes it (lines 14 and 15)
   - 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

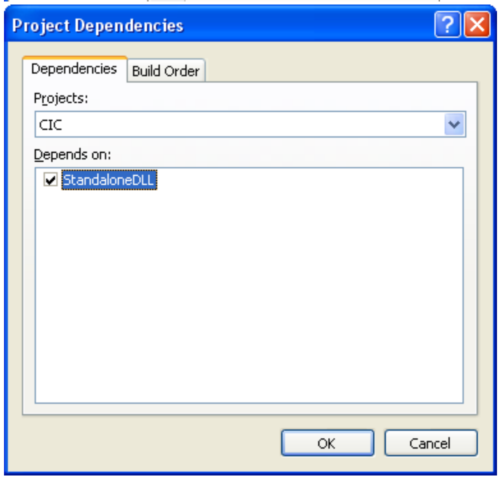
- 
CodeGenExample - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Debug
Win32
DimCarrierIndexEnum
Solution Explorer - Solution 'CodeGen...
Solution 'CodeGenExample' (2 projects)
CIC
Header Files
stdafx.h
targetver.h
Resource Files
Source Files
CIC.cpp
stdafx.cpp
ReadMe.txt
StandaloneDLL
CIC.cpp
Source Control Explorer
(Global Scope)

```
// 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;
}
```

```
// 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.

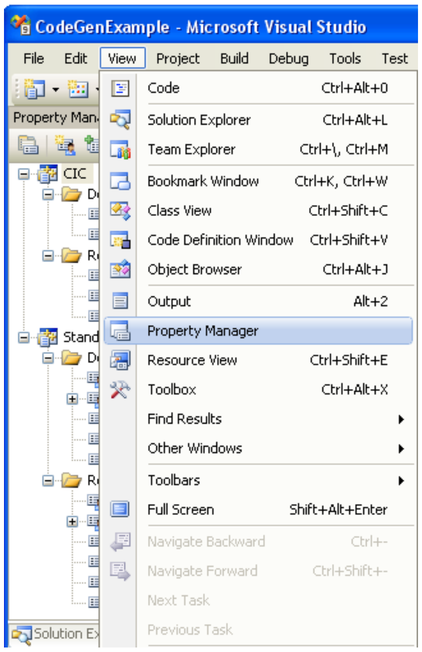
- 

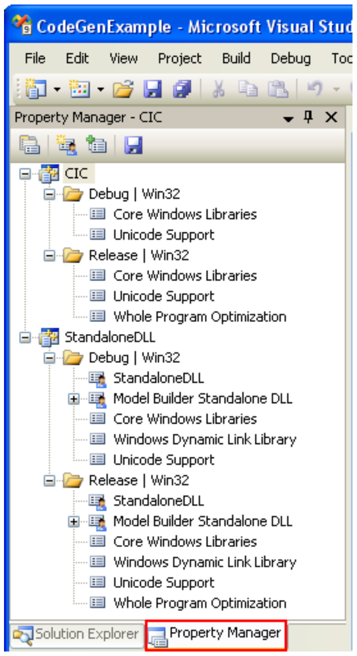


Projects: CIC

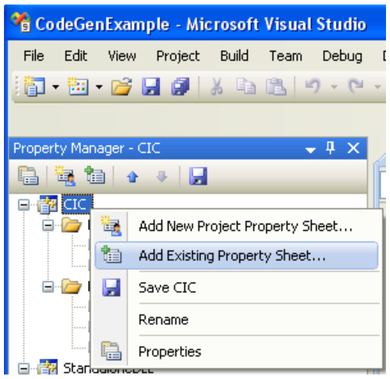
Depends on:
- [✓] StandaloneDLL

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

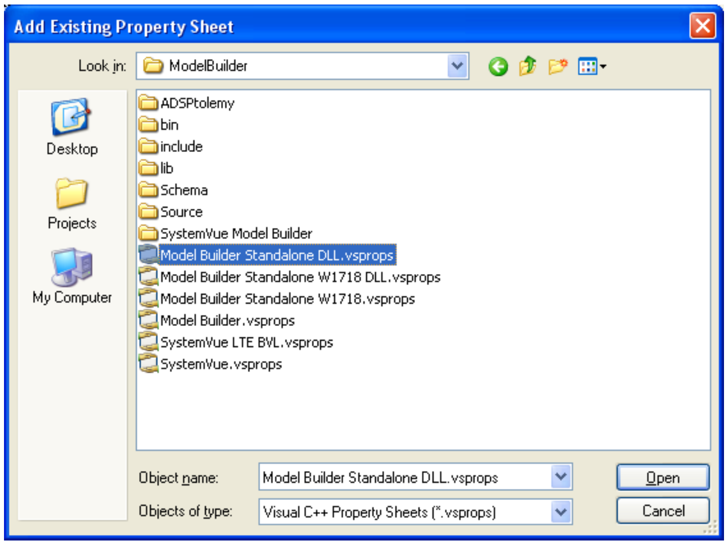

CodeGenExample - Microsoft Visual Studio
File Edit View Project Build Debug Tools Test
Code Ctrl+Alt+0
Solution Explorer Ctrl+Alt+L
Team Explorer Ctrl+\, Ctrl+M
Bookmark Window Ctrl+K, Ctrl+W
Class View Ctrl+Shift+C
Code Definition Window Ctrl+Shift+V
Object Browser Ctrl+Alt+J
Output Alt+2
Property Manager
Resource View Ctrl+Shift+E
Toolbox Ctrl+Alt+X
Find Results
Other Windows
Toolbars
Full Screen Shift+Alt+Enter
Navigate Backward Ctrl+-
Navigate Forward Ctrl+Shift+-
Next Task
Previous Task
Property Man
CIC
Stand
Solution E

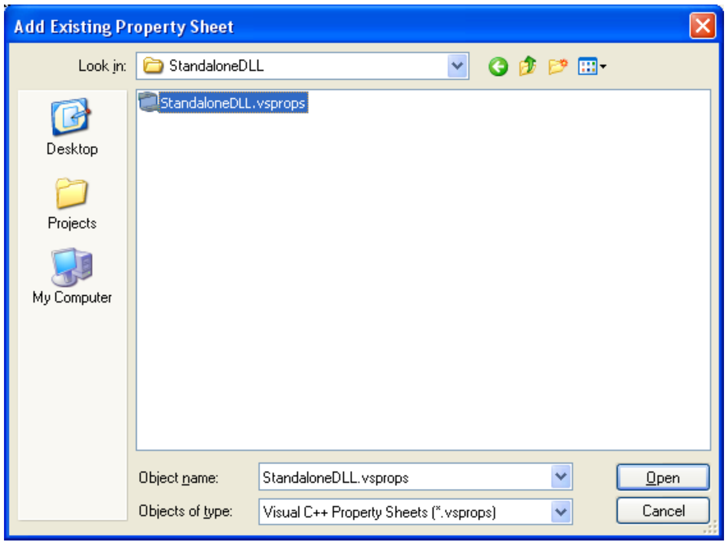


   - Add the "Model Builder Standalone DLL" property sheet to the CIC project. To do this right click on the CIC project and select **Add Existing Property Sheet...**.

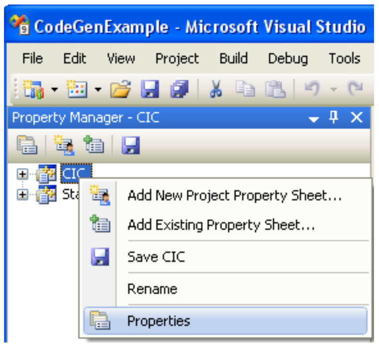
- 
CodeGenExample - Microsoft Visual Studio
File Edit View Project Build Team Debug
Property Manager - CIC
CIC
Add New Project Property Sheet...
Add Existing Property Sheet...
Save CIC
Rename
Properties

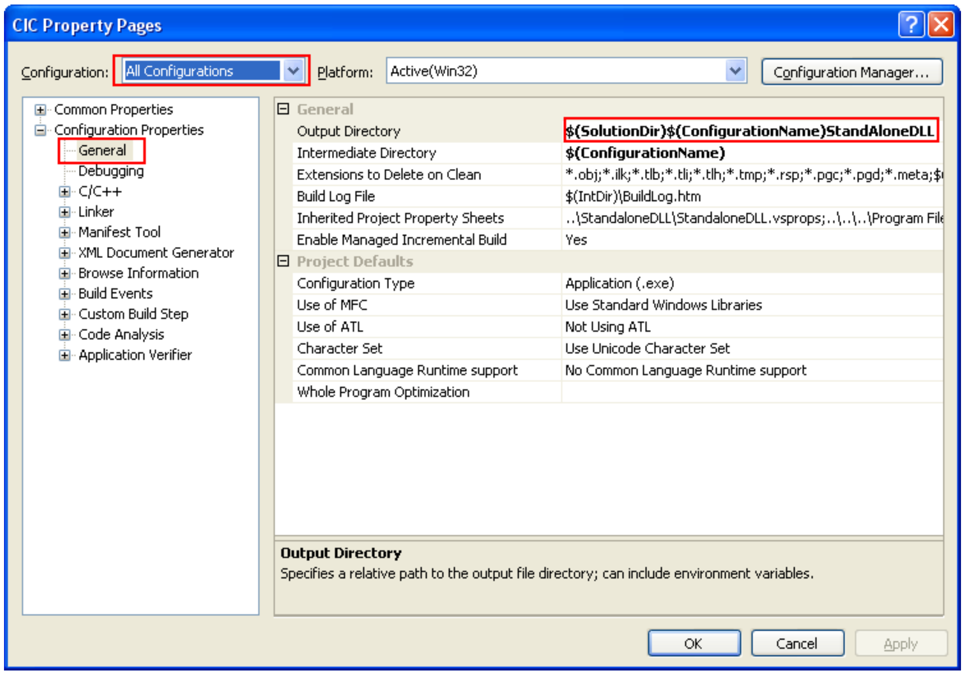
  Navigate to the Modelbuilder directory of your SystemVue installation, select "Model Builder Standalone DLL.vsprops", and press the open button.

  
Add Existing Property Sheet
Look in: ModelBuilder
Desktop
Projects
My Computer
ADSPtolemy
bin
include
lib
Schema
Source
SystemVue Model Builder
Model Builder Standalone DLL.vsprops
Model Builder Standalone W1718 DLL.vsprops
Model Builder Standalone W1718.vsprops
Model Builder.vsprops
SystemVue LTE BVL.vsprops
SystemVue.vsprops
Object name: Model Builder Standalone DLL.vsprops
Objects of type: Visual C++ Property Sheets (*.vsprops)
Open
Cancel
- 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).

  
Add Existing Property Sheet
Look in: StandaloneDLL
Desktop
Projects
My Computer
StandaloneDLL.vsprops
Object name: StandaloneDLL.vsprops
Objects of type: Visual C++ Property Sheets (*.vsprops)
Open
Cancel

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**.

- 
CodeGenExample - Microsoft Visual Studio
File Edit View Project Build Debug Tools
Property Manager - CIC
CIC
Add New Project Property Sheet...
Add Existing Property Sheet...
Save CIC
Rename
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.

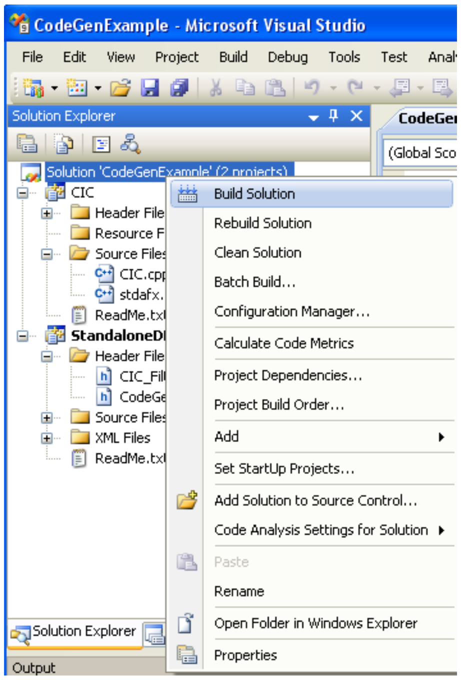


Configuration: All Configurations Platform: Active(Win32)

General	
Output Directory	$(SolutionDir)$(ConfigurationName)StandAloneDLL
Intermediate Directory	$(ConfigurationName)
Extensions to Delete on Clean	*.obj;*.ilk;*.tlb;*.tli;*.tlh;*.tmp;*.rsp;*.pgc;*.pgd;*.meta;$
Build Log File	$(IntDir)\BuildLog.htm
Inherited Project Property Sheets	..\StandaloneDLL\StandaloneDLL.vsprops;..\..\..\Program File
Enable Managed Incremental Build	Yes
Project Defaults	
Configuration Type	Application (.exe)
Use of MFC	Use Standard Windows Libraries
Use of ATL	Not Using ATL
Character Set	Use Unicode Character Set
Common Language Runtime support	No Common Language Runtime support
Whole Program Optimization	

Output Directory
Specifies a relative path to the output file directory; can include environment variables.

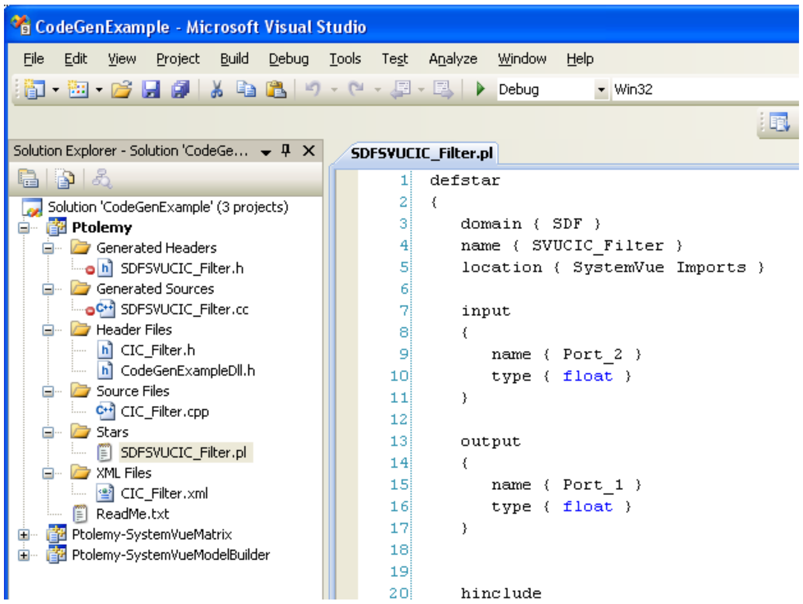
10. Build the solution by right clicking on the solution name and selecting **Build Solution**.



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 shell 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.



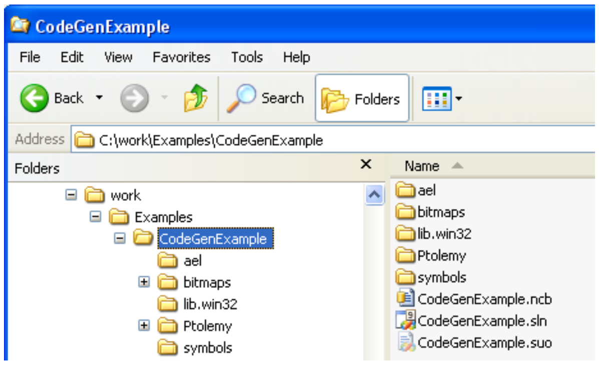
```
defstar
{
   domain { SDF }
   name { SVUCIC_Filter }
   location { SystemVue Imports }

   input
   {
      name { Port_2 }
      type { float }
   }

   output
   {
      name { Port_1 }
      type { float }
   }


   hinclude
```

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 shell 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** shell. 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\SystemVue2010.07
   - 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\SystemVue2010.07\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\SystemVue2010.07\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 (shell type).

SystemVue Model Shell

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 Shell

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 Shell

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\2010.07\W1718" under "My Documents" directory for your user account, where 2010.07 represents corresponding SystemVue version. 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\2010.07\LTE_8.9" under "My Documents" directory for your user account, where 2010.07 represents corresponding SystemVue version, and 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.

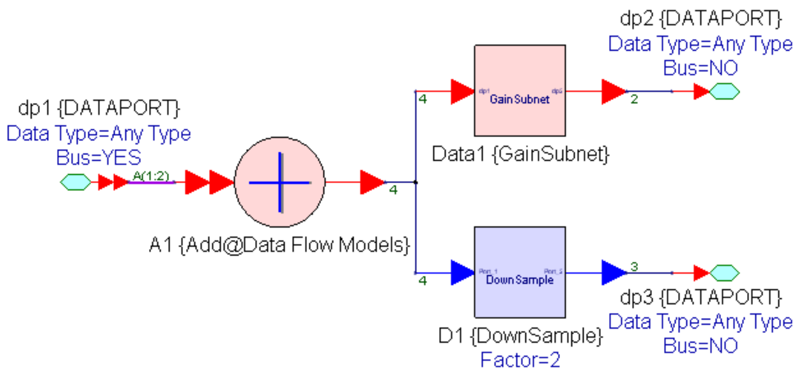
  ℹ Model's inputs, outputs, parameters, array parameter sizes, and enum types must be declared in public scope, and the names and enum types must be specified properly.

- After code generation, the Visual Studio solution and projects (see Generating Code and Supported Shells) 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.

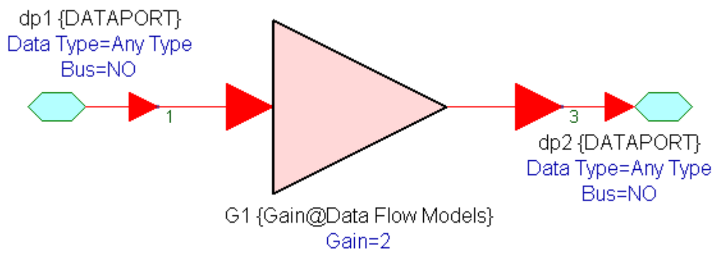
Understanding Generated C++ Code

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.

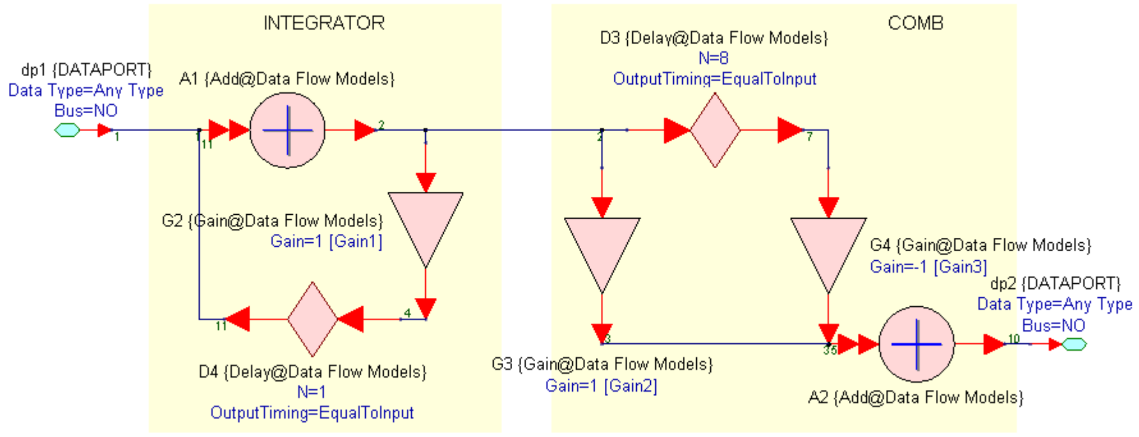
Original Model	**Generated Model**
Math (algorithm)	performs only the function selected during code generation (FunctionType parameter is removed)
MathCx (algorithm)	performs only the function selected during code generation (FunctionType parameter is removed)
Trig (algorithm)	performs only the function 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)

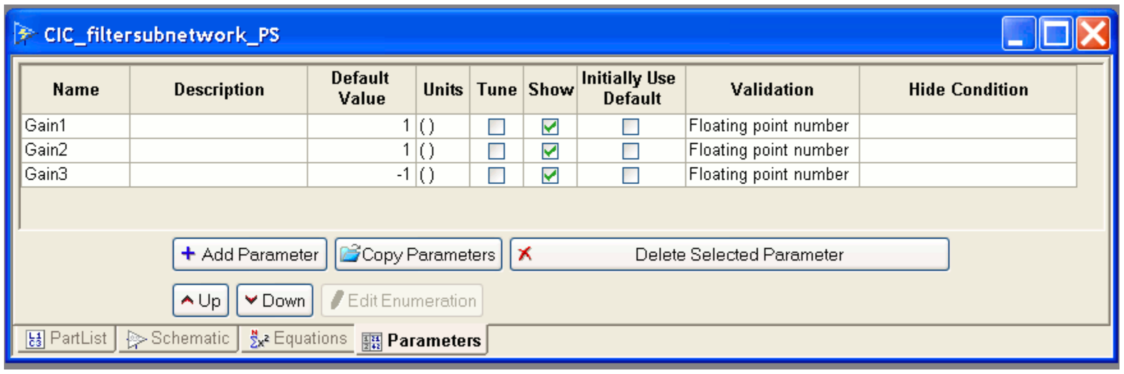
3. See next section on Parameter Support for other cases where the generated code may not behave the same as the SystemVue sub-network it was generated from.

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*.


INTEGRATOR
COMB
dp1 {DATAPORT}
Data Type=Any Type
Bus=NO
A1 {Add@Data Flow Models}
D3 {Delay@Data Flow Models}
N=8
OutputTiming=EqualToInput
G2 {Gain@Data Flow Models}
Gain=1 [Gain1]
G4 {Gain@Data Flow Models}
Gain=-1 [Gain3]
dp2 {DATAPORT}
Data Type=Any Type
Bus=NO
D4 {Delay@Data Flow Models}
N=1
OutputTiming=EqualToInput
G3 {Gain@Data Flow Models}
Gain=1 [Gain2]
A2 {Add@Data Flow Models}



Name	Description	Default Value	Units	Tune	Show	Initially Use Default	Validation	Hide Condition
Gain1		1	()	☐	☑	☐	Floating point number	
Gain2		1	()	☐	☑	☐	Floating point number	
Gain3		-1	()	☐	☑	☐	Floating point number	

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.

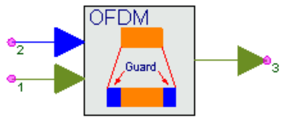
AddGuard Part

Categories: *C++ Code Generation* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AddGuard (algorithm)	OFDM Symbol Guard Samples Inserter

AddGuard (OFDM Symbol Guard Samples Inserter)



Description: OFDM Symbol Guard Samples Inserter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *AddGuard Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
IFFTSize	IFFT size	64		Integer	NO
PreGuard	Pre-guard length	16		Integer	NO
PostGuard	Post-guard length	0		Integer	NO
Intersection	Guard intersection length	0		Integer	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	In	Transmitted signal after IFFT	complex	NO
2	Window	Window function	real	NO

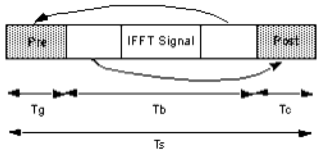
Output Ports

Port	Name	Description	Signal Type	Optional
3	Out	OFDM output data	complex	NO

Notes/Equations

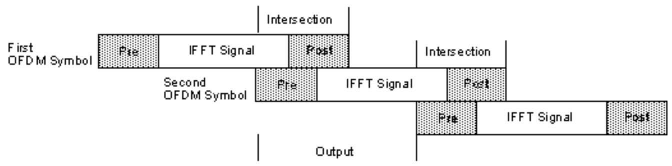
1. Guard intervals are added to IFFT signals to form an OFDM symbol. Both Pre- and post-guard intervals are possible.
2. IFFTSize specifies the input IFFT signal length.
3. PreGuard specifies the pre-guard length. If PreGuard = 0, there is no pre-guard. PostGuard specifies the post-guard length. If PostGuard = 0, there is no post-guard.
4. Intersection specifies the intersect length of two consecutive OFDM symbols. If Intersection = 0, there is no intersect between symbols. To protect the IFFT signals, Intersection cannot exceed PreGuard + PostGuard.
5. IEEE 802 series (802.11a, 802.11g, 802.15.3a, 802.16a, 802.16d) and DVB-T standards do not include post-guard and intersection.
6. For each OFDM symbol:
   - IFFTSize samples are input from pin In.
   - PreGuard + IFFTSize + PostGuard samples are input from pin Window.
   - PreGuard + IFFTSize + PostGuard - Intersection samples are output as the OFDM symbol.
7. A window function is input at pin Window for multiplying the output OFDM symbol. A non-window is created by providing a constant one at this pin.
8. If an intersect does not exist, the windowed OFDM symbol is output.
9. If an intersect does exist, the windowed OFDM symbol and the intersect of the previous OFDM symbol is added before output. The next intersect of the windowed OFDM symbol is saved for the next OFDM symbol.
10. How an OFDM symbol is formed.
    - Inverse-Fourier-transformation creates the IFFT signal of time duration Tb (having IFFTSize samples) and constitute the useful symbol.
    - A copy of the last time duration Tg (having PreGuard samples) of the useful symbol is added before the IFFT signal. This pre-guard is also called a cyclic prefix.
    - A copy of the first time duration Tc (having PostGuard samples) of the useful symbol is added after the IFFT signal. This post-guard is also called a cyclic postfix.
    - The combined duration is referred to as symbol time Ts. OFDM Symbol Time with Guard Interval illustrates this sequence.

OFDM Symbol Time with Guard Interval



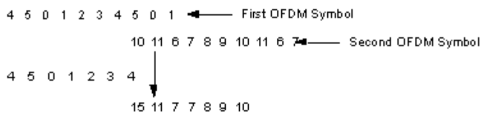
11. How Intersection, PreGuard and PostGuard values form consecutive OFDM symbols for Case 1: Intersection > PreGuard and Intersection > PostGuard.
    - The second OFDM symbol is formed and multiplied by window.
    - Intersect of the first and second OFDM symbols is then summed and output first.
    - Beginning after Intersection samples, the remaining segment of the second OFDM symbol with length of PreGuard + IFFTSize + PostGuard - 2 × Intersection is output.
    - Last Intersection samples of the second OFDM symbol are saved as for the next OFDM symbol.

    Intersection > PreGuard, Intersection > PostGuard

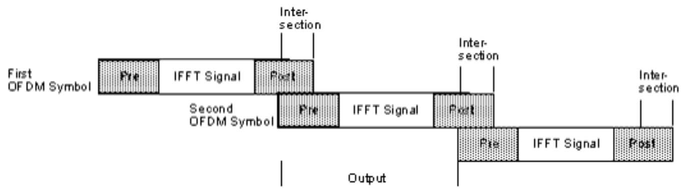
    Let the input be {0, 1, 2, 3, 4, 5} and {6, 7, 8, 9, 10, 11}, window is 1, IFFTSize = 6, PreGuard = 2, PostGuard = 2, Intersection = 3. With the steps described above, the output of the first and second OFDM symbol are {4, 5, 0, 1, 2, 3, 4} and {15, 11, 7, 7, 8, 9, 10}, respectively. Case 1: Calculation for Output illustrates the calculation.

    Case 1: Calculation for Output

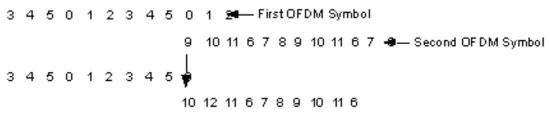
12. How Intersection, PreGuard and PostGuard values form consecutive OFDM symbols for Case 2: Intersection ≤ PreGuard and Intersection ≤ PostGuard.
    - This calculation is similar to Case 1.

    Intersection ≤ PreGuard, Intersection ≤ PostGuard

    Let the input be {0, 1, 2, 3, 4, 5} and {6, 7, 8, 9, 10, 11}, window is 1, IFFTSize = 6, PreGuard = 3, PostGuard = 3, Intersection = 2. The output of the first and second OFDM symbols are {3, 4, 5, 0, 1, 2, 3, 4, 5, 0} and {10, 12, 11, 6, 7, 8, 9, 10, 11, 6}, respectively. Case 2: Calculation for Output illustrates the calculation.

    Case 2: Calculation for Output

References

1. IEEE Standard 802.11a-1999, "Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: High-speed Physical Layer in the 5 GHz Band," 1999.
2. ETSI TS 101 475 v1.1.1, "Broadband Radio Access Networks (BRAN); HIPERLAN Type 2; Physical (PHY) layer," April, 2000.
3. ARIB-JAPAN, Terrestrial Integrated Services Digital Broadcasting (ISDB-T);

Specification of Channel Coding, Frame Structure and Modulation, Sept.1998.
4. ETSI, Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for digital terrestrial television. EN300 744 v1.2.1, European Telecommunication Standard, July 1999.
5. IEEE P802.15-03/268r1, "Multi-band OFDM Physical Layer Proposal for IEEE 802.15 Task Group 3a," September 2003.
6. IEEE P802.16-REVd/D2-2003, "Draft IEEE Standard for Local and metropolitan area networks Part 16: Air Interface for Fixed Broadband Wireless Access Systems," 2003.

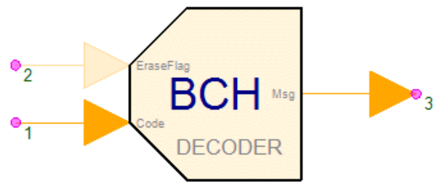
BCH_Decoder Part

Categories: *C++ Code Generation* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
BCH_Decoder (algorithm)	Binary primitive BCH decoder

BCH_Decoder



Description: Binary primitive BCH decoder
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *BCH Decoder Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
M	The root of generation polynomial is defined in GF(2^M), primitive codeword length N=2^M-1	3		Integer	NO
K	Primitive message length (Unshortened)	4		Integer	NO
T	Error correction capability	1		Integer	NO
CodeLength	Shortened codeword length (<=2^M-1), set 0 for unshortened code	0		Integer	NO
PrimPoly	Primitive polynomial in the form of integer (default []), binary vector or power vector of non-zero item	[]		Integer array	NO
Erase	there is erased bits or not in undecoded code: NO, YES	NO		Enumeration	NO
ErasePosition	index array of erased bits (from [0,CodeLength-1]), valid if Erase==YES and pin EraseFlag is disconnected	[]		Integer array	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	Code	Uncoded binary codeword	int	NO
2	EraseFlag	Specifying the corresponding bits is erased (1) or not (0)	int	YES

Output Ports

Port	Name	Description	Signal Type	Optional
3	Msg	Decoded binary messageword	int	NO

Notes/Equations

1. This model performs the decoding of primitive binary BCH (Bose-Chaudhuri-Hocquengham) systematic code.
2. The primitive message length, is represented by **K**, and the primitive codeword length, 2^M-1, is represented by **N**. For shortened code, **Ks** and **Ns** represent the shortened message length and codeword length respectively.
3. Each run, it consumes Ns = min(N,CodeLength) input codeword bits and output Ks = K+min(0,CodeLength-2^M-1) decoded message bits. CodeLength=0 is treated the same as CodeLength=N.
4. The primitive polynomial of the Galois Field GF(2^M), p(x) = 1+p(2)*x+p(3)*x^2 +...+p(M)*x^{M-1}+x^M, is set by array PrimPoly in three ways. If p(x)= 1+x+x^4, i.e. p(x) = 1*x^0+1*x^1+0*x^2+0*x^3+1*x^4, PrimPoly can be

   an array consists of the order of x whose coefficients are 1, i.e.

   [0,1,4]

   an array consists of all polynomial coefficients, from the lowest power to the highest power, i.e.

   [1,1,0,0,1]

   an integer p(2), i.e.

   19

   Note
   If PrimPoly is set to [], the default primitive polynomials shown below shall be used
5. The relationship of T and code generation polynomial g(x)

   T is the error correction capability, i.e. the number of reducible error bits in the received codeword.

2*T+1 equals to the minimum distance between different codewords, or the number of 1's in the coefficient array of g(x).

6. Decoding algorithm
   - Constructing the codeword polynomial

     $r(x) = r(0)*x^{Ns-1} + r(1)*x^{Ns-2} + ... + r(Ns-2)*x + r(Ns-1)$

     from un-decoded input (r(0) is the first input bit)

     r = [r(0), r(1), ..., r(Ns-1)]
   - Calculating the syndrome **S**

     $S(i) = r(\alpha^i)$, i=1,2,...,T*2, α is a primitive element of $GF(2^M)$

     If {S(i)} are all zeros, i.e. no (reducible) errors, picking up [r(0), r(1), ..., r(Ks-1)] as the decoded message and return. Otherwise, going forward to the next step
   - Calculating error location polynomial with Berlekamp-Massey algorithm

     $\sigma(x) = \sigma(0) + \sigma(1)*x + \sigma(2)*x^2 + ... + \sigma(v)*x^v, v \le T$

     by filling the iterative table

u	**$\sigma^u(x)$**	**d_u**	**l_u**	**$u-l_u$**
-1	1	1	0	-1
0	1	S_1	0	0
1				
2				
...				
2T				

     with the following iteration method

     Assuming the result of the u^th iteration is

     $\sigma^u(x) = \sigma^u(0) + \sigma^u(1)*x + \sigma^u(2)*x^2 + ... + \sigma^u(l_u)*x^{lu}$

     to deduce $\sigma^{u+1}(x)$, calculate the u^th discrepancy

     $d_u = S_{u+1} + \sigma^u(1)S_u + \sigma^u(2)S_{u-1} + ... + \sigma^u(l_u)S_{u+1-lu}$

     If d_u==0,

     $\sigma^{u+1}(x) = \sigma^u(x)$

     otherwise, search $\sigma^\rho(x)$ satisfying ρ<u, $d_\rho \neq 0$ and $\rho-l_\rho=\max\{i-l_i\}$, -1≤i<u, and rewrite $\sigma^{u+1}(x)$ as

     $\sigma^{u+1}(x) = \sigma^u(x) + d_u\, d_\rho^{-1}\, x^{u-\rho}\, \sigma^\rho(x)$
   - Calculating the error locations with Chien Search

     Create a all-zero array **e** = [e(0), e(1), ..., e(Ns-1)]

     Evaluate σ(x) with $\{\alpha^{i+1}$, N-Ns≤i<N}. If $\sigma(\alpha^i)$==0, i.e. r(i-(N-Ns)) is a error bit, set e(i) to 1.
   - Correcting error bits with error array **e** obtained above and get the decoded output bits

     $\mathbf{u_{dec}}$ = [r(0), r(1), ..., r(Ks-1)] + [e(0), e(1), ..., e(Ks-1)]
7. If Erase is YES, the decoder shall perform a decoding with erased bits.
   - The positions of erased bits is set by ErasePosition consisting of not more than 2T digits with each in the range of [0,CodeLength)
   - If the EraseFlag pin is connected, parameter ErasePosition shall be ignored. EraseFlag should has the same length as Input and in which 1 represents an erased bits and 0 represents not.
   - The decoder decode the codeword twice by replacing the erased bits with logic 1's and 0's respectively. Decoded message is obtained from one of the two decoded codewords which has the smaller distance from the uncoded codewords (excluding the erased positions).
8. Default primitive polynomials

M	Primitive Polynomial	Integer representation
1	p(x) = x + 1	3
2	p(x) = x^2 + x + 1	7
3	p(x) = x^3 + x + 1	11
4	p(x) = x^4 + x + 1	19
5	p(x) = x^5 + x^2 + 1	37
6	p(x) = x^6 + x + 1	67
7	p(x) = x^7 + x^3 + 1	137
8	p(x) = x^8 + x^4 + x^3 + x^2 + 1	285
9	p(x) = x^9 + x^4 + 1	529
10	p(x) = x^{10} + x^3 + 1	1033
11	p(x) = x^{11} + x^2 + 1	2053
12	p(x) = x^{12} + x^6 + x^4 + x + 1	4179
13	p(x) = x^{13} + x^4 + x^3 + x + 1	8219
14	p(x) = x^{14} + x^{10} + x^6 + x + 1	17475
15	p(x) = x^{15} + x + 1	32771
16	p(x) = x^{16} + x^{12} + x^3 + x + 1	69643
17	p(x) = x^{17} + x^3 + 1	131081
18	p(x) = x^{18} + x^7 + 1	262273
19	p(x) = x^{19} + x^5 + x^2 + x + 1	524327
20	p(x) = x^{20} + x^3 + 1	1048585

References

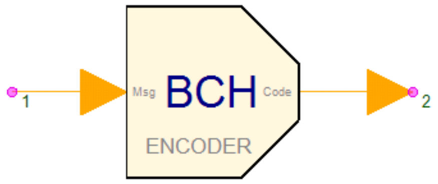
BCH_Encoder Part

Categories: *C++ Code Generation* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
BCH_Encoder (algorithm)	Binary BCH Encoder

BCH_Encoder



Description: Binary BCH Encoder
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *BCH Encoder Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
M	The root of generation polynomial is defined in GF(2^M), primitive codeword length N=2^M-1	3		Integer	NO
K	Primitive message length (Unshortened)	4		Integer	NO
MsgLength	Shortened message length [0,K], set 0 (or K) for unshortened code	0		Integer	NO
GenPoly	Code generation polynomial (if g(x)=1+X+X^3, GenPoly=[0,1,3] or [1,1,0,1])	[0,1,3]		Integer array	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	Msg	Uncoded binary message	int	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Code	Encoded binary codeword	int	NO

Notes/Equations

1. This model performs the encoding of binary BCH (Bose-Chaudhuri-Hocquengham) systematic code.
2. The primitive message length, is represented by **K**, and the primitive codeword length, 2^M-1, is represented by **N**. For shortened code, **Ks** and **Ns** represent the shortened message length and codeword length respectively.
3. Each run, it consumes Ks = min(K,MsgLength) input message bits and output Ns = N+min(0,MsgLength-K) codeword bits. MsgLength=0 is treated the same as MsgLength=K.
4. The code generation polynomial, g(x), with the highest order x^{N-K}, is set by array GenPoly in three ways.
   - 1-D array of the order of x whose coefficients are 1. For instance, if g(x)= $1+x+x^3$, i.e. g(x) = $1*x^0+1*x^1+0*x^2+1*x^3$, GenPoly should be [0,1,3]. The number of parity bits is the highest power of g(x).
   - 1-D array consists of all polynomial coefficients, from lowest power to highest power. If g(x)= $1+x+x^3$, i.e. g(x) = $1*x^0+1*x^1+0*x^2+1*x^3$, GenPoly should be [1,1,0,1].
   - 2-D array consists of the polynomial coefficients of g(x)'s factor polynomials whose highest power ≤M. For instance, if g(x) = $(1+x+x^4) * (1+x+x^2+x^3+x^4) * (1+x+x^2)$, GenPoly can be set as a 3 x (M+1) array

     [1,1,0,0,1;
     1,1,1,1,1;
     1,1,1,0,0]
5. Coding process
   - Constructing the message polynomial

     $u(x) = u(0)*x^{Ks-1} + u(1)*x^{Ks-2} + ... + u(Ks-2)*x + u(Ks-1)$

     from input Msg (rewrite as u, u(0) is the first input bit)

     u = [u(0), u(1), ..., u(Ks-1)]
   - Dividing $x^{Ns-Ks}u(x)$ by g(x) and obtaining the remainer polynomial

     $b(x) = x^{Ns-Ks}u(x)/g(x) = b(0)*x^{Ns-Ks-1} + b(1)*x^{Ns-Ks-2} + ... + b(Ns-Ks-2)*x + b(Ns-Ks-1)$
   - Combining u(x) and g(x) to obtain the codeword polynomial c(x) (c(0) is the first output bit)

$c(x) = x^{Ns-Ks}u(x) + b(x) = c(0)*x^{Ns-1} + c(1)*x^{Ns-2} + ... + c(Ns-2)*x + c(Ns-1)$

and the codeword array

c = [c(0), c(1), ... c(Ns-1)] = [u(0), u(1), ... u(Ks-1), b(0), b(1), ..., b(Ns-Ks-1)]

References

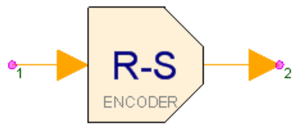
CoderRS Part

Categories: *C++ Code Generation* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
CoderRS (algorithm)	Reed Solomon Encoder

CoderRS (Reed Solomon Encoder)



Description: Reed Solomon Encoder
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *CoderRS Part* (algorithm)

Model Parameters

Name	**Description**	**Default**	**Units**	**Type**	**Runtime Tunable**
GF	Galois field size (2^GF)	8		Integer	NO
CodeLength	Length of output codeword	255		Integer	NO
MessageLength	Length of input message symbols	223		Integer	NO
PrimPoly	Coefficients of primitive polynomial. PrimPoly must be the coefficients of the m order of polynomial	[1, 0, 1, 1, 1, 0, 0, 0, 1]		Integer array	NO
Root	First root of generator polynomial	1		Integer	NO

Input Ports

Port	**Name**	**Description**	**Signal Type**	**Optional**
1	in	information symbol	int	NO

Output Ports

Port	**Name**	**Description**	**Signal Type**	**Optional**
2	out	systematical code	int	NO

Notes/Equations

1. The CoderRS model implements a Reed-Solomon (RS) encoder.
2. This model reads k samples from input *in* and writes n samples to output *out*.
3. RS codes are a class of block codes that operate on non-binary symbols. The symbols are formed from *m* bits of a binary data stream. A code block is then formed with $n = 2^m - 1$ symbols. In each block, *k* symbols are formed from the encoder input and ($n - k$) parity symbols are added. The code is thus a systematic code. The rate of the code is k/n, and the code is able to correct up to $t = (n - k - 1)/2$ or $(n - k)/2$ symbol errors in a block, depending on whether $n - k$ is odd or even. For example, the code used in the WCDMA [1] data transmission system is a (36,32) code shortened from RS code (255,251) defined on Galois Field (2^8). A shortened code can be formed by taking 32 input symbols, padding them out with 219 all zero symbols to form 251 symbols, and then encoding with a RS code (255,251). The 219 fixed symbols are then discarded prior to transmission. The input pin consumes k tokens and the output pin produces n tokens for each firing.
4. Implementation
   The code format is: RS code (n, k), defined on Galois Field (2^m).
   Galois Field Generator
   Galois Fields are set up according to the number of bits per symbol and the number of symbols per block.
   Generate GF (2^m) from the irreducible primitive polynomial. It is defined as the polynomial of least degree, with coefficients in GF(2) and a highest degree coefficient equal to 1. The polynomial is always of degree *m*.
   The elements of Galois Field can have two representations: exponent or polynomial.
   Let α represent the root of the primitive polynomial p(x). Then in GF(2^m), for any $0 \le i \le 2^m - 2$
   $$\alpha^i = b_i(0) + b_i(1)\alpha + b_i(2)\alpha^2 + \cdots + b_i(m-1)\alpha^{m-1}$$
   where the binary vector (bi(0), bi(1),..., bi(m_-1)) is the representation of the integer polynomial[i]. Now exponent[i] is the element whose polynomial representation is (bi(0), bi(1),..., bi(m-1)), and exponent[polynomial[i]] = i.

Polynomial representation is convenient for addition, exponent representation for multiplication.

RS Encoder

The RS generator polynomial is generally defined as

$$g(x) = (x - a^{m_0})(x - a^{m_0 + 1})\ldots(x - a^{m_0 + 2t - 1})$$

where t is the correctable error number. It can be reduced to a 2t order of polynomial

$$g(x) = x^{2t} + g_{2t-1}x^{2t-1} + \ldots + g_0$$

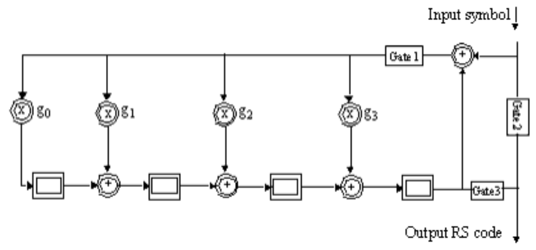
Encoding is done by using a feedback shift register with appropriate connections specified by the element g $_i$. The encoded symbol is then

$$in(x) \times x^{(n-k)} + parity(x)$$

where in(x) is the polynomial representation of the input data, parity(x) is the polynomial of the parity symbol.

The RS encoder diagram is illustrated in Reed Solomon Encoder.

Figure: Reed Solomon Encoder



See Also:

DecoderRS (algorithm)

References

1. NTT Mobile Communications Network Inc. "Specifications for W-CDMA Mobile Communication System Experiment", October 9, 1997.
2. S. Lin, D. J. Costello, Error Control Coding Fundamentals and Applications, 1983.

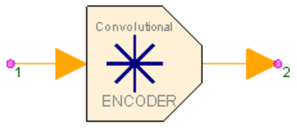
ConvolutionalCoder Part

Categories: *C++ Code Generation* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
ConvolutionalCoder (algorithm)	Convolutional Coder

ConvolutionalCoder (ConvolutionalCoder)



Description: Convolutional Coder
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *ConvolutionalCoder Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
CodingRate	Coding rate: rate_1_2, rate_1_3, rate_1_4, rate_1_5, rate_1_6, rate_1_7, rate_1_8	rate_1_2		Enumeration	NO
ConstraintLength	Constraint length	7		Integer	NO
Polynomial	Generator polynomial	[91, 121]		Integer array	NO
ZeroTail	Zero tail used to convert convolutional code to block code: NO, YES	NO		Enumeration	NO
BitSequenceLength	Length of bit squence not including tail bits	88		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	In	boolean	NO

Output Ports

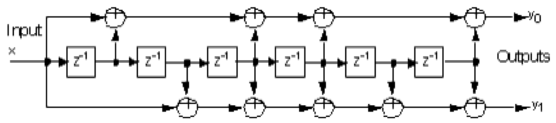
Port	Name	Signal Type	Optional
2	Out	boolean	NO

Notes/Equations

1. The ConvolutionalCoder model convolutionally encodes the input information sequence bit-by-bit.
2. This model reads 1 sample from input *In* and writes R samples to output *Out*. R = 2, 3, 4, 5, 6, 7, or 8 when CodingRate = rate_1_2, rate_1_3, rate_1_4, rate_1_5, rate_1_6, rate_1_7, or rate_1_8 respectively.
3. A convolutional code is generated by passing the information sequence to be transmitted through a linear finite-state shift register. The shift register generally consists of K(*k*-bit) stages and *n* linear algebraic function generators. Input data to the encoder (assumed to be binary) is shifted into (and along) the shift register *k* bits at a time. The number of output bits for each *k*-bit input sequence is *n* bits. Therefore, the code rate is defined as $R_c = k/n$, which is consistent with the code rate definition for a block code. The K parameter is called the constraint length of the convolutional code.
4. CodingRate (R_c) is the ratio of input bit (*k*) and output bits (*n*). ConvolutionalCoder supports the 1/*n* coding rate only, which implements an $R_c = 1/n$ rate(n = 2, 3, 4, 5, 6, 7, 8) convolution for input data.
   Convolutional codes with k/n ($k > 1$) are not supported by this component because: coding and decoding will be more complex; and, even convolutional codes with a k/n ($k > 1$) coding rate are used that are typically implemented by puncturing the convolutional code with a 1/*n* coding rate.
5. ConstraintLength (*K*) represents shift register stages.
6. Polynomial is the generator function of the convolutional code. In general, the generator matrix for a convolutional code is semi-infinite since the input sequence is semi-infinite. As an alternative to specifying the generator matrix, a functionally equivalent representation is used in which a set of n vectors is specified, one vector for each *n* modulo-2 adder. A 1 in the ith position of the vector indicates that the corresponding stage in the shift register is connected to the modulo-2 adder; 0 in a given position indicates that no connection exists between that stage and the modulo-2 adder.

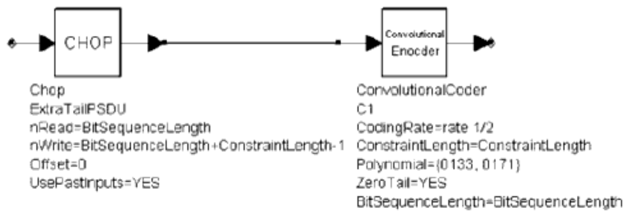
For example, consider the binary convolutional encoder with constraint length *K* = 7, *k* = 1, and *n* = 2; refer to Convolutional Code CC(2, 1 ,7). The connection for y0 is (1, 0, 1, 1, 0, 1, 1) from Outputs to Input; the connection for y1 is (1, 1, 1, 1, 1, 0, 1). The generators for this code are more conveniently given in octal form as (0133, 0175). So, when *k* = 1, *n* generators, each of dimension *K* specify the encoder.

Convolutional Code CC(2, 1 ,7)



7. ZeroTail specifies the character of encoder input sequence. If ZeroTail = YES, the input sequence of encoder is divided into blocks. The length of the block is BitSequenceLength. After each block, *K* - 1 zeros need to be appended as tail bits. That is, the total block length of encoder is (BitSequenceLength + *K* - 1), referring to Tail bits appending for ZeroTail = YES. The information will be used in the decoder to obtain better performance.

Tail bits appending for ZeroTail = YES



Chop
ExtraTailPSDU
nRead=BitSequenceLength
nWrite=BitSequenceLength+ConstraintLength-1
Offset=0
UsePastInputs=YES

ConvolutionalCoder
C1
CodingRate=rate 1/2
ConstraintLength=ConstraintLength
Polynomial=(0133, 0171)
ZeroTail=YES
BitSequenceLength=BitSequenceLength

8. BitSequenceLength (valid only if ZeroTail = YES) is used to specify the information bit length, which indicates the length of uncoded bits. This parameter can be used to set the same value for the encoder and the decoder.

See Also:

ViterbiDecoder (algorithm)

References

1. John G. Proakis, Digital Communications (Third edition), Publishing House of Electronics Industry, Beijing, 1998.

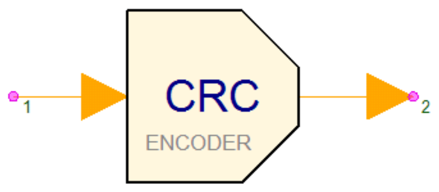
CRC_Coder Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
CRC_Coder (algorithm)	CRC Coder

CRC_Coder (CRC Coder)



Description: CRC Coder
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *CRC Coder Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
ParityPosition	Parity bits position: Tail, Head	Tail		Enumeration	NO	
ReverseData	Reverse the data sequence: NO, YES	NO		Enumeration	NO	
ReverseParity	Peverse the parity bits: NO, YES	NO		Enumeration	NO	
ComplementParity	Complement parity bits: NO, YES	NO		Enumeration	NO	
MessageLength	Input message length	172		Integer	NO	[1:∞)
InitialState	Initial state of encoder	0		Integer	NO	[0:∞)
Polynomial	Generator polynomial	7955		Integer	NO	[3:∞)

Input Ports

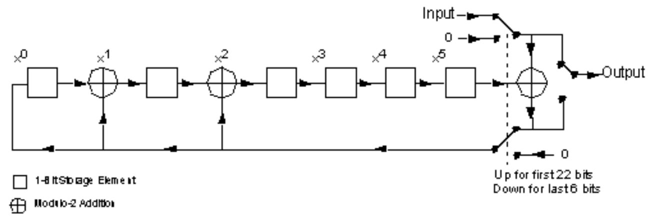
Port	Name	Description	Signal Type	Optional
1	In	input data	boolean	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Out	output data	boolean	NO

Notes/Equations

1. This component is used to add CRC bits to the input information. Each firing, (MessageLength + CRCLength) tokens are produced when MessageLength tokens are consumed. CRCLength is the length of CRC bits that is determined by Polynomial, where $2^{CRCLength} \le$ Polynomial $\le 2^{CRCLength+1}$.
2. CRC bits can be added to the head or the tail of the information bits by setting ParityPosition. The order of CRC bits and the order of information bits can be reversed by setting ReverseData and ReverseParity.
3. CRC Bit Calculation as shown below is an example of a CRC encoder in CDMA2000, where $g(x) = x^6 + x^2 + x + 1$, and Polynomial is hex 0x47. The CRC bits are added after the information bits; the order of the CRC and information bits are not reversed.



- Initially, all shift register elements are set to the InitialState and the switches are set in the up position.
- The register is clocked the number of times equal to MessageLength.
- Switches are then set in the down position so that the output is a modulo-2 addition with a 0 and the successive shift register inputs are 0.
- The register is clocked an additional number of times equal to CRCLength and the CRC bits are output.

References

1. TIA/EIA/IS-2000.2 (PN-4428), Physical Layer Standard for cdma2000 Spread Spectrum Systems, July 1999.

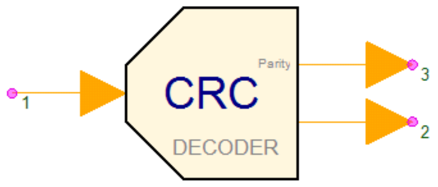
CRC_Decoder Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
CRC_Decoder (algorithm)	CRC Decoder

CRC_Decoder (CRC Decoder)



Description: CRC Decoder
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *CRC Decoder Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
ParityPosition	Parity bits position: Tail, Head	Tail		Enumeration	NO	
ReverseData	Reverse the data sequence: NO, YES	NO		Enumeration	NO	
ReverseParity	Peverse the parity bits: NO, YES	NO		Enumeration	NO	
ComplementParity	Complement parity bits: NO, YES	NO		Enumeration	NO	
MessageLength	Input message length	172		Integer	NO	[1:∞)
InitialState	Initial state of encoder	0		Integer	NO	[0:∞)
Polynomial	Generator polynomial	7955		Integer	NO	[3:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	In	input data	boolean	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Out	output data	boolean	NO
3	Parity	Parity check	int	NO

Notes/Equations

1. This component is used to check the CRC bits for CRC frame errors.
   Each firing, (MessageLength + CRCLength) tokens are consumed when MessageLength tokens and one parity token are produced. CRCLength is the CRC bit length determined by Polynomial, where $2^{CRCLength} \le$ Polynomial $\le 2^{CRCLength+1}$.
2. The message part of the input data is sent to a CRC encoder that has the same Polynomial value as the encoder (CRC_Coder). The CRC bits are then compared with the CRC bits in the input data. If these are the same, the CRC check is passed.

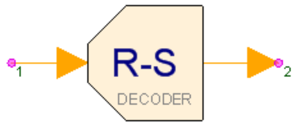
DecoderRS Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
DecoderRS (algorithm)	Reed Solomon Decoder

DecoderRS (Reed Solomon Decoder)



Description: Reed Solomon Decoder
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *DecoderRS Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
GF	Galois field size (2^GF)	8		Integer	NO	[2:30]	m
CodeLength	Length of input codewords	255		Integer	NO	[3:2<sup>m</sup>-1]	n
MessageLength	Length of output message symbols	223		Integer	NO	[1:CodeLength-2]	k
PrimPoly	Coefficients of primitive polynomial	[1, 0, 1, 1, 1, 0, 0, 0, 1]		Integer array	NO	†	p
Root	First root of generator polynomial	1		Integer	NO	[0:2<sup>m</sup>-1 - (n - k)]	m_0

Input Ports

Port	Name	Description	Signal Type	Optional
1	in	received symbol	int	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	out	decoded symbol	int	NO

Notes/Equations

1. The DecoderRS model implements a Reed-Solomon (RS) decoder.
2. This model reads n samples from input *in* and writes k samples to output *out*.
3. The RS decoding is performed via the Berlekamp iterative algorithm [2].
4. The Berlekamp iterative algorithm locates the error in RS code and generates an error location polynomial. By finding the root of the error location polynomial, the error position can be determined. If decoding is successful, the information symbols are output; otherwise, the received data is unaltered.
5. Decoding routines are described here.
   For the shortened code, the same number of symbols 0 is inserted into the same position as CoderRS and a Reed Solomon decoder is used to decode the block. After decoding, the padded symbols are discarded, leaving the desired information symbols.
   Syndromes indicate erroneous situations. When the generator polynomial g(x) and the received codeword represented by r(x) are given, one or more errors have occurred during transmission of an encoded block.
   Let
   $$v(x) = v_0 + v_1 x + v_2 x^2 + \ldots + v_{n-1} x^{n-1}$$
   where v(x) is the polynomial representation of the transmitted symbol.
   $$r(x) = r_0 + r_1 x + r_2 x^2 + \ldots + r_{n-1} x^{n-1}$$
   where r(x) is the polynomial representation of the received symbol.
   Then
   $$r(x) = v(x) + e(x)$$
   where e(x) denotes the error patterns.
   If r_i - v_i, then e_i = 0; else e_i = 1.

Remember that

v(*x*) = *g*(*x*)*Q*(*x*)

where Q(x) is the quotient.

So if α^i is the root of g(x) , then $v(\alpha^i) = 0$ and $r(\alpha^i) = e(\alpha^i)$.

Now there is a simple procedure for checking the occurrence of errors at the receiver:

Calculate syndromes *s*(*i*), the syndromes are decided by the error patterns:

$$s(i) = e(\alpha^{m_0 + i})$$

If one or more of the syndromes are not equal to zero, one or more symbol errors occur in the received data. For example, if

$$\alpha^{m_0}, \alpha^{m_0+1}, \ldots, \alpha^{m_0} + 2^{t-1}$$

are roots of g(x), then

$$s(1) = r(\alpha^{m_0})$$

$$s(1) = r(\alpha^{m_0+1})$$

.

.

.

$$s(2t) = r(\alpha^{m_0+2t-1})$$

Syndromes are used to find the error location polynomial.

Given the syndromes s(i), the decoding algorithm will synthesize an error location polynomial. The roots of the polynomial indicate the error positions.

Assuming the received symbols have v symbol errors, the syndromes are represented as:

$$s(1) = \beta_1^{m_0} + \beta_2^{m_0} + \ldots + \beta_v^{m_0}$$

$$s(2) = \beta_1^{m_0+1} + \beta_2^{m_0+1} + \ldots \beta_v^{m_0+1}$$

.

.

.

$$s(2t) = \beta_1^{m_0+2t-1} + \beta_2^{m_0+2t-1} + \ldots \beta_v^{m_0+2t-1}$$

where the error location is

$$\beta_l = a^{i_l}$$

and

$$a^{i_l}(1 \le l \le v)$$

Now the error location polynomial is defined as

$$\Omega(x) = \left(1 + \beta_1^{m_0}x\right)\left(1 + \beta_2^{m_0}x\right)\ldots\left(1 + \beta_v^{m_0}x\right)$$

$$= \Omega_0 + \Omega_1 x + \Omega_2 x^2 + \ldots + \Omega_v x^v$$

The Berlekamp iterative algorithm is used to construct this polynomial, which is the key to RS decoding.

The algorithm is described here without proof; for more information, see Ref. [1].

An iterative table will be filled.

μ	$\Omega^{(u)}(x)$	d_μ	l_μ	$\mu - l_\mu$
-1	1	1	0	-1
0	1	s_1	0	0
1				
2 ... , 2t				

where

μ

is the iterative step number

d_μ

is the mth step iterative difference

l_μ

is the order of $\Omega^{(\mu)}(x)$

If

$d_\mu = 0$

then

$$\Omega^{(\mu+1)}(x) = \Omega_\mu^{(\mu)}(x)$$

and

$$l_{\mu+1} = l_\mu$$

If

$d_\mu \neq 0$

search for lines in the table to find step *p* in which *d* $_p$ ≠ 0 and the value of *p* - *l* $_p$ is the maximum, then

$$\Omega^{(\mu+1)}(x) = \Omega^{(\mu)}(x) - d_\mu d_\rho^{-1} x^{(\mu-\rho)} \Omega^{(\rho)}(x)$$

and

$$l_{\mu+1} = max(l_\mu, l_\rho + \mu - \rho)$$

For the two conditions

$$d_{\mu+1} = s_{\mu+2} + \Omega_1^{(\mu+1)} s_{\mu+1} + \ldots + \Omega_{l_\mu+1}^{(\mu+1)} s_{\mu+2-l_{\mu+1}}$$

Iterate until the last line of the table Ω $^{(2\ t)}$ (*x*) is calculated. If the order of the polynomial is greater than *t* (which means the received codeword block has more than *t* errors) the error cannot be corrected.
For non-binary codes, the error values must be known.
The minimum order polynomial is iteratively solved to obtain the least number of roots (error location number). The inverse element of the root indicates the error location.
The error value is calculated based on the Ref. [2] equation

$$e_{jl} = \beta_l^{(1-m_0)} \frac{z(\beta_l^{-1})}{\prod_{\substack{i=1 \\ i \neq l}}^{v} (1 + \beta_i \beta_l^{-1})}$$

where

$$z(x) = 1 + (s_1 + \Omega_1)x + (s_2 + \Omega_1 s_1 + \Omega_2)x^2 + \\ \ldots + (s_v + \Omega_1 s_{v-1} + \Omega_2 s_{v-2} + \ldots + \Omega_v)x^v$$

Then,

$$out(x) = r(x) - e(x)$$

See Also:

CoderRS (algorithm)

References

1. E.R. Berlekamp, Algebraic Coding Theory, McGraw-Hill, New York, 1968.
2. S. Lin, D. J. Costello, Error Control Coding Fundamentals and Applications, 1983.

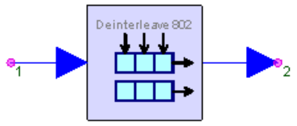
Deinterleaver802 Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Deinterleaver802 (algorithm)	IEEE 802 Deinterleaver

Deinterleaver802 (IEEE 802 Deinterleaver)



Description: IEEE 802 Deinterleaver
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Deinterleaver802 Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
s	Modular factor of interleaving	1		Integer	NO	[1:∞)
l	Divisor factor of interleaving	16		Integer	NO	[1:∞)
NCBPS	Number of coded bits per OFDM symbol	48		Integer	NO	[1:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	In	Input	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Out	Output	real	NO

Notes/Equations

1. The Deinterleaver802 model performs deinterleaving based on IEEE 802 standards. It performs the inverse of the Interleaver802 model.
2. This model reads N_{CBPS} samples from input *In* and writes N_{CBPS} samples to the output *Out*.
   - N_{CBPS} is the number of coded bits in a single OFDM symbol and is defined in the table below.
3. Deinterleaving is defined by a two-step permutation; *j* is used to denote the index of the original received bit before the first permutation; *i* is used to denote the index after the first (and before the second) permutation; *k* is used to denote the index after the second permutation, before delivering the coded bits to the convolutional (Viterbi) decoder.
   The first permutation is defined by
   $i = s \times \text{floor}(j/s) + (j + \text{floor}(l \times j/ N_{CBPS})) \bmod s$ $j = 0, 1, \ldots N_{CBPS} - 1$
   The function floor (.) denotes the largest integer not exceeding the parameter
   The second permutation is defined by
   $k = l \times i - (N_{CBPS} - 1)\text{floor}(l \times i/N_{CBPS})$ $i = 0, 1, \ldots N_{CBPS} - 1$
   In the equations, *s is the modular factor and l is the divisor factor*; these are variable parameters and their values depend on which standard the model is used for.
   If this model is used for IEEE 802.11 and HIPERLAN/2
   $s = \max(N_{BPSC}/2, 1)$, $l = 16$
   where
   N_{BPSC} and N_{CBPS} are determined by data rates given in IEEE 802.11 and HIPERLAN/2 Rate Dependent Values.
   If this model is used for IEEE 802.16
   $s = N_{BPSC}/2, 1)$ $l = 12$
   where N_{BPSC} and N_{CBPS} are determined by block sizes given in IEEE 802.16 Bit Interleaver Block Sizes (NCBPS / NBPSC).

IEEE 802.11 and HIPERLAN/2 Rate Dependent Values

Data Rate (Mbps)	Modulation	Coding Rate (R)	Coded Bits per Subcarrier (NBPSC)	Coded Bits per OFDM Symbol (NCBPS)	Data Bits per OFDM Symbol (NDBPS)
6	BPSK	1/2	1	48	24
9	BPSK	3/4	1	48	36
12	QPSK	1/2	2	96	48
18	QPSK	3/4	2	96	72
24 (IEEE 802.11a)	16QAM	1/2	4	192	96
27 (HIPERLAN/2)	16QAM	9/16	4	192	108
36	16QAM	3/4	4	192	144
48 (IEEE 802.11a)	64QAM	2/3	6	288	192
54	64QAM	3/4	6	288	216

IEEE 802.16 Bit Interleaver Block Sizes (N_{CBPS} / N_{BPSC})

Modulation	16 Subchannels (Default)	8 Subchannels	4 Subchannels	2 Subchannels	1 Subchannel
QPSK	384/2	192/2	96/2	48/2	24/2
16QAM	768/4	384/4	192/4	96/4	48/4
64QAM	1152/6	576/6	288/6	144/6	72/6

See Also:

Interleaver802 (algorithm)

References

1. IEEE Standard 802.11a-1999, "Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: High-speed Physical Layer in the 5 GHz Band," 1999.
2. ETSI TS 101 475 v1.1.1, "Broadband Radio Access Networks (BRAN); HIPERLAN Type 2; Physical (PHY) layer," April, 2000.
3. IEEE P802.16-REVd/D2-2003," Part 16 Air Interface for Fixed Broadcast Wireless Access Systems".

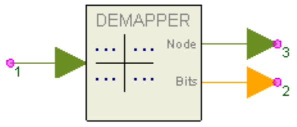
Demapper Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Demapper (algorithm)	Complex Symbol Demapper/Slicer

Demapper (Complex Symbol Demapper)



Description: Complex Symbol Demapper/Slicer
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Demapper Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
ModType	Modulation type: BPSK, QPSK, PSK8, PSK16, QAM16, QAM32, QAM64, QAM128, QAM256, User_Defined	QPSK		Enumeration	NO
MappingTable	Constellation table	[1, -1]		Complex array	NO
BitOrder	Bit order: LSB first, MSB first	LSB first		Enumeration	YES

Input Ports

Port	Name	Description	Signal Type	Optional
1	In	input symbol sequence	complex	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Bits	output bit sequence	boolean	NO
3	Node	closest constellation node	complex	NO

Notes/Equations

1. Demapper inputs a complex value, finds the nearest constellation node for the input, and outputs both the constellation node and the symbol value for the constellation node in a bit sequence specified by the BitOrder parameter.
2. For each input, one constellation node is output at Node and depending on the ModType parameter Symbol Length number of bits is output at Bits.
3. A constellation value is a pair of real values (I,Q) that is expressed on the input as I + jQ. Earlier in the modulation chain, I modulated the inphase part of the carrier, and Q modulated the quadrature part of the carrier over a symbol period.
4. Each modulation type has its constellation and symbol length. The symbol length, i.e. the number of output bits per symbol, is detailed in the following table.

   Modulation Type and Symbol Length

ModType	Symbol Length
BPSK	1
QPSK	2
PSK8	3
PSK16 or *QAM16*	4
QAM32	5
QAM64	6
QAM128	7
QAM256	8

   If ModType is *User_Defined* and the size of MappingTable is N, then the symbol length is $\log_2$ (N) bits.
5. For ModType *BPSK*, bit value 0 is mapped to 1 + j0 and bit value 1 is mapped to -1 + j0.
6. For ModType *QPSK*, the constellation map is illustrated in QPSK Modulation

Constellation. For ModType *PSK8*, the constellation map is illustrated in 8PSK Modulation Constellation. For ModType *PSK16*, the constellation map is illustrated in 16PSK Modulation Constellation. The output symbols are assumed to be Gray coded.

7. The symbol mappings for ModType *QAM16*, *QAM32*, *QAM64*, *QAM128* and *QAM256* are described in the section 9 of [1], and their constellation maps are illustrated in figure 7-8 of [1].
8. QAM constellations need definition only for quadrant 1. The constellation points in quadrants 2, 3 and 4 are derived from quadrant 1 by selecting the quadrant 1 constellation value with the least significant bits of the desired symbol and rotating that constellation value by the amount selected by the two most significant bits of the desired symbol, $b_i\ b_q$, as specified in table Conversion of Constellation Points.

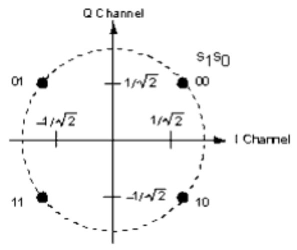
Conversion of Constellation Points

Quadrant	Symbol Most Significant Bits ($b_i\ b_q$)	Rotation
1	00	0
2	10	п/2
3	11	п
4	01	3п/2

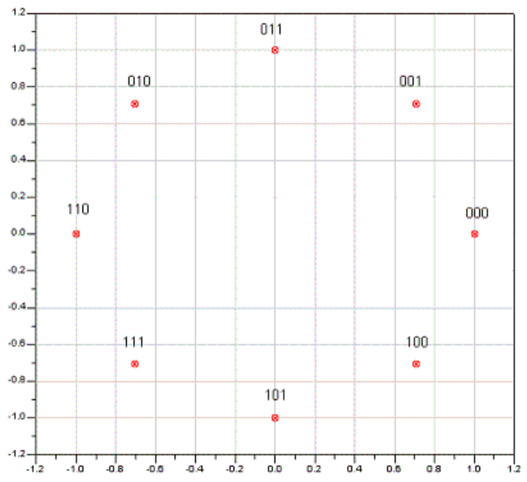
16QAM, 32QAM, 64QAM, 128QAM and 256QAM constellation maps are illustrated in 16 and 32QAM Constellation through 256QAM Constellation.

9. When ModType is specified to *User_Defined*, a custom constellation is defined with MappingTable. The output symbol is mapped directly to a constellation point as a 0 based index into MappingTable.

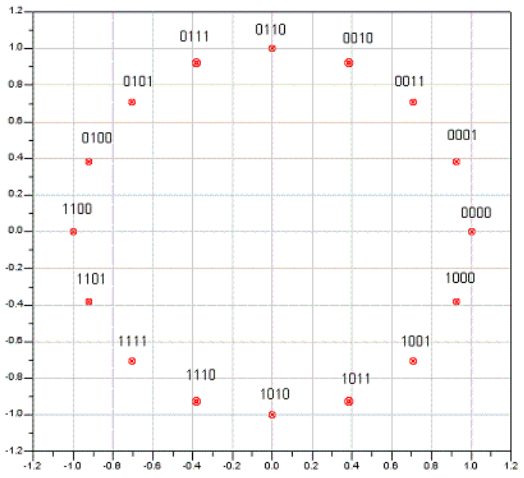
QPSK Modulation Constellation



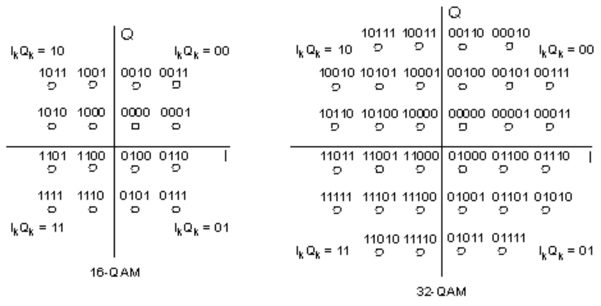
8PSK Modulation Constellation



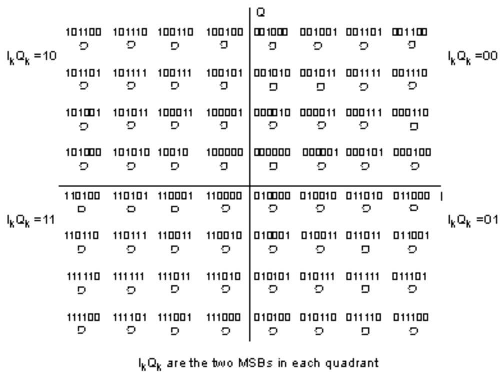
16PSK Modulation Constellation



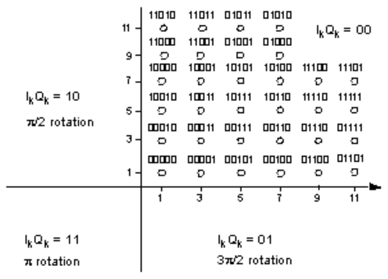
16 and 32QAM Constellations



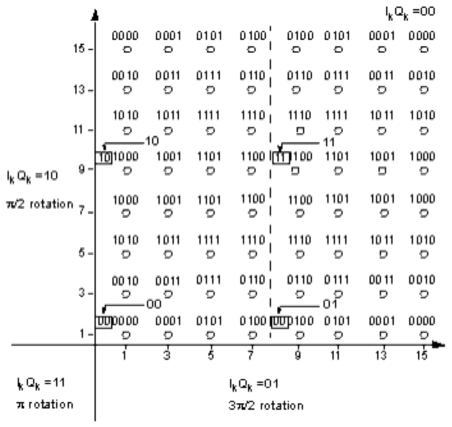
64QAM Constellation



128QAM Constellation



256QAM Constellation



See *Mapper* (algorithm).

References

1. EN 300 429, "Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for cable systems," V1.2.1, 1998-04.

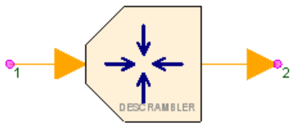
DeScrambler Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
DeScrambler (algorithm)	Bit Sequence Descrambler

DeScrambler (Bit Sequence Descrambler)



Description: Bit Sequence Descrambler
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *DeScrambler Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Polynomial	Generator polynomial for the shift register - decimal, octal, or hex integer	147457		Integer	NO	(0:∞)
ShiftReg	Initial state of the shift register - decimal, octal, or hex integer	1		Integer	NO	(-∞:∞)

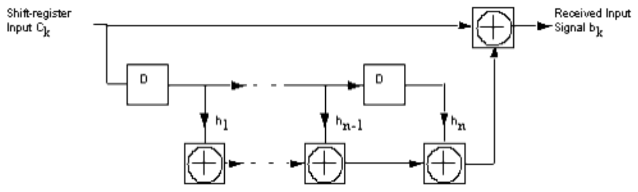
Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input bit sequence (zero or nonzero)	boolean	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output bit sequence (zero or one)	boolean	NO

Notes/Equations

1. This component descrambles the input bit sequence using a feedback shift register. The taps of the feedback shift register are given by the Polynomial parameter. This is a self-synchronizing descrambler that will exactly reverse the operation of the Scrambler component if the corresponding parameter values of Scrambler and DeScrambler are the same.
   A self-synchronized descrambler is shown in *Self-Synchronized Descrambler*.
   Figure:Self-Synchronized Descrambler
   

2. See also, *Scrambler* (algorithm).

References

1. E. A. Lee and D. G. Messerschmitt, *Digital Communication*, Second Edition, Kluwer Academic Publishers, 1994, pp. 595-603.

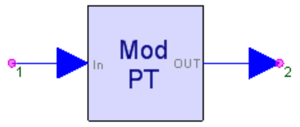
FMPulseTrain Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
FMPulseTrain (algorithm)	Frequency modulated pulse train

FMPulseTrain



Description: Frequency modulated pulse train
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *FMPulseTrain Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Frequency	Nominal frequency of the pulse train	20e3	Hz	Float	NO	(0:∞)
InitialPhase	Initial phase	0	deg	Float	YES	(-∞:∞)
FreqSensitivity	Frequency deviation sensitivity in Hz/Volt	1e5		Float	YES	(-∞:∞)
Amplitude	Amplitude of the pulse train	1	V	Float	YES	(-∞:∞)
PulseWidth	Time width of the output pulse train	1e-6	s	Float	YES	(0:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output	real	NO

Notes/Equations

1. FMPulseTrain generates a frequency modulated pulse train.
2. For every input, there is one output.
3. Consider a phasor P(t) operating at $P'(t) = 2\pi\{Frequency + FrequencySensitivity \times input(t)\}$ radians per second.
4. P(t) is P(0) + $\int_0^t P'(\lambda)d\lambda$

   where P(0) is InitialPhase.
5. If P(t) crosses $2\pi n$ for integer n within the last time step, a pulse is generated.
6. A pulse has Amplitude value and persists PulseWidth seconds. No pulse is represented by a zero output.
7. PulseWidth is rounded to the nearest time step and is forced to be at least one time step.
8. Pulses may overlap, however the most recently generated pulse determines the next no pulse time.

See also: *Modulator* (algorithm).

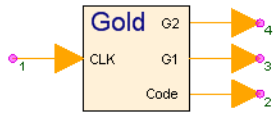
GoldCode Part

Categories: *C++ Code Generation* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
GoldCode (algorithm)	Gold Code Generator

GoldCode



Description: Gold Code Generator
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *GoldCode Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
GoldCodePolynomials	G1,G2 polynomials for LFSR-1,LFSR-2: G1[5 2]&G2[5 4 3 2], G1[7 3]&G2[7 3 2 1], G1[7 3 2 1]&G2[7 5 4 3 2 1], G1[8 7 6 5 2 1]&G2[8 7 6 1], G1[9 4]&G2[9 6 4 3], G1[9 6 4 3]&G2[9 8 4 1], G1[10 3]&G2[10 9 8 6 3 2]_GPS, G1[10 9 8 7 6 5 4 3]&G2[10 9 7 6 4 1], G1[10 8 7 6 5 4 3]&G2[10 9 7 6 4 1], G1[10 8 5 1]&G2[10 7 6 4 2 1], G1[10 9 8 6 5 1]&G2[10 5 3 2], G1[10 8 4 3]&G2[10 9 6 5 4 3], G1[10 8 7 6 2 1]&G2[10 9 4 2], G1[10 5 2 1]&G2[10 9 7 6 4 3 2 1], G1[11 2]&G2[11 8 5 2], G1[11 8 5 2]&G2[11 10 3 2], G1[11 7 3 2]&G2[11 10 9 7 6 4 3 2], G1[12 9 3 2]&G2[12 11 8 7 6 3 2 1], G1[12 11 6 4 2 1]&G2[12 11 8 7 5 4 3 2], G1[12 9 8 3 2 1]&G2[12 11 8 7 3 1], G1[12 9 8 5 4 3]&G2[12 7 6 4], G1[12 10 7 5 3 2]&G2[12 11 8 4 3 1], G1[12 11 7 4 2 1]&G2[12 10 9 8 4 3], G1[12 10 6 5 2 1]&G2[12 11 10 4], G1[12 11 10 9 8 7 5 4 3 2]&G2[12 11 7 4], G1[12 9 7 6 3 1]&G2[12 11 10 7 6 5 4 1], G1[12 11 10 9 4 2]&G2[12 8 4 3 2 1], G1[12 9 8 5 4 3 2 1]&G2[12 11 7 5 4 3], G1[12 10 8 7 6 2]&G2[12 10 2 1], G1[13 4 3 1]&G2[13 12 8 7 6 5], G1[13 10 9 7 5 4]&G2[13 12 8 7 6 5], G1[13 11 8 7 4 1]&G2[13 11 10 5 4 3 2 1]	G1[10 3]&G2[10 9 8 6 3 2]_GPS		Enumeration	NO
G1InitialState	LFSR-1 initial state non-zero bit positions	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]		None	NO
G2InitialState	LFSR-2 initial state non-zero bit positions	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]		None	NO
PhaseSelect	LFSR-2 output modification only for GPS Goldcode: PRN_disabled, PRN1_2xor6, PRN2_3xor7, PRN3_4xor8, PRN4_5xor9, PRN5_1xor9, PRN6_2xor10, PRN7_1xor8, PRN8_2xor9, PRN9_3xor10, PRN10_2xor3, PRN11_3xor4, PRN12_5xor6, PRN13_6xor7, PRN14_7xor8, PRN15_8xor9, PRN16_9xor10, PRN17_1xor4, PRN18_2xor5, PRN19_3xor6, PRN20_4xor7, PRN21_5xor8, PRN22_6xor9, PRN23_1xor3, PRN24_4xor6, PRN25_5xor7, PRN26_6xor8, PRN27_7xor9, PRN28_8xor10, PRN29_1xor6, PRN30_2xor7, PRN31_3xor8, PRN32_4xor9, PRN33_5xor10, PRN34_4xor10, PRN35_1xor7, PRN36_2xor8, PRN37_4xor10	PRN_disabled		Enumeration	NO
SkipNBits	Number of bits to initially skip	0		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	clock	int	NO
2	reset	int	YES

Output Ports

Port	Name	Signal Type	Optional
3	goldcode	int	NO
4	g1	int	NO
5	g2	int	NO

Notes/Equations

1. Gold code sequences have the property of minimal circular autocorrelation values for nonzero lag.
2. For every clock input, there is a Gold code output.
3. New bit values are output when the clock input is true (not 0). Otherwise, the output bit value is unchanged.
4. Gold code sequences are generated with a pair of linear feed back register (*LFSR* (algorithm)) of the same length. These registers named G1 and G2 have very specific feedback taps (polynomials) which are selected through the GoldCodePolynomials parameter as a pair of polynomials. The GoldCodePolynomials enumeration G1[5 2]&G2[5 4 3 2] define a 5 bit LFSR for both G1 and G2. G1 has feedback taps at bit positions 5 and 2, and G2 has feedback taps at bit position 5, 4, 3 and 2.
5. G1 and G2 output are both found at their LFSR highest bit position, i.e. if G1 and G2 are defined with GoldCodePolynomials enumeration G1[5 2]&G2[5 4 3 2], then G1 and G2 output from bit position 5. The outputs are typically exclusive ORed to generate a Gold code bit. When clocked G1 and G2 are shifted one bit simultaneously toward the higher bit position to generate the next Gold code bit.
6. G1InitialState and G2InitialState initializes G1 and G2 respectively. Both parameters can be specified either as a bit array or as an array of positions having bit value 1. The implied length of the bit array notation is the number of array elements. For the positional notation, the implied length is the largest position value.
7. PhaseSelect is displayed only when GoldCodePolynomials is G1[10 3]&G2[10 9 9 6 3 2]_GPS. This Gold code polynomial is used in Global Positioning Satellite (GPS) hardware. PhaseSelect chooses two bits from G2 which are together exclusive ORed with the G1 output to produce the Gold code bit. The enumeration PRN1_2xor9 choose bit position 2 and 9 from G2.
8. SkipNBits truncates the Gold code sequence by removing the first SkipNBits number of outputs from each Gold code sequence period.

See:
Bits
LFSR (algorithm)
SetSampleRate (algorithm)

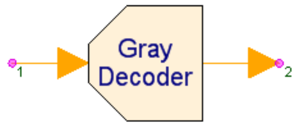
GrayDecoder Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
GrayDecoder (algorithm)	Gray Decoder

GrayDecoder (Gray Decoder)



Description: Gray Decoder
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *GrayDecoder Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NumBits	Number of bits read/produced per execution	4		Integer	NO	[1:∞)
BitOrder	Bit order: LSB first, MSB first	MSB first		Enumeration	NO	

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input bits	boolean	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output bits	boolean	NO

Notes/Equations

1. This model implements a Gray decoder.
2. At every execution NumBits bits are read from the input, Gray decoded, and written to the output.
3. The BitOrder parameter specifies whether the first bit in the block of NumBits bits read/written is to be considered as MSB or LSB.
4. See also: *GrayEncoder* (algorithm)

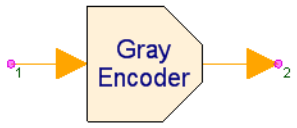
GrayEncoder Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
GrayEncoder (algorithm)	Gray Encoder

GrayEncoder (Gray Encoder)



Description: Gray Encoder
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *GrayEncoder Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NumBits	Number of bits read/produced per execution	4		Integer	NO	[1:∞)
BitOrder	Bit order: LSB first, MSB first	MSB first		Enumeration	NO	

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input bits	boolean	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output bits	boolean	NO

Notes/Equations

1. This model implements a Gray encoder.
2. At every execution NumBits bits are read from the input, Gray encoded, and written to the output.
3. The BitOrder parameter specifies whether the first bit in the block of NumBits bits read/written is to be considered as MSB or LSB.
4. Gray encoding maps natural binary numbers to a set of binary codes where two successive codes differ only in one bit.
   1-bit binary numbers and their Gray codes

Binary Number	Gray Code
0	0
1	1

2-bit binary numbers and their Gray codes (MSBs are to the left and LSBs to the right)

Binary Number	Gray Code
00	00
01	01
10	11
11	10

3-bit binary numbers and their Gray codes (MSBs are to the left and LSBs to the right)

Binary Number	Gray Code
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

4-bit binary numbers and their Gray codes (MSBs are to the left and LSBs to the right)

Binary Number	Gray Code
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

The easiest way to construct a (N+1)-bit Gray code is to start with the N-bit Gray code, reflect it, then prepend the top half with 0 and the bottom half with 1. Following is an example showing how the 3-bit Gray code is constructed from the 2-bit Gray code.

- Start with 2-bit Gray code
  00
  01
  11
  10
- Reflect
  00
  01
  11
  10

  10
  11
  01
  00
- Prepend 0 for the top half and 1 for the bottom half
  000
  001
  011
  010

  110
  111
  101
  100

5. Gray encoding is commonly used in communications systems when mapping bits to constellation points so that adjacent constellation points are assigned bit patterns that differ only in one bit. This helps minimize Bit Error Rate since in most cases errors occur between adjacent constellation points and in this case only one bit will be in error.
6. See also: *GrayDecoder* (algorithm)

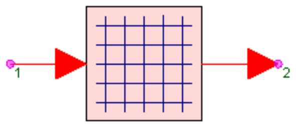
InterleaveDeinterleave Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
InterleaveDeinterleave (algorithm)	Interleaver / Deinterleaver

InterleaveDeinterleave (Interleaver/Deinterleaver)



Description: Interleaver / Deinterleaver
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *InterleaveDeinterleave Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Rows	Number of rows of the interleave/deinterleave matrix	8		Integer	NO	(0:∞)
Columns	Number of columns of the interleave/deinterleave matrix	8		Integer	NO	(0:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Notes/Equations

1. This component is a general purpose interleaver/de-interleaver. Every time it fires it reads (Rows × Columns) samples from its input and writes them to its output in a different order. Its operation is equivalent to writing the samples read from its input in a Rows × Columns matrix row-wise, then reading the matrix elements column-wise and writing them to the output.
   Alternatively, the *Transpose* (algorithm) component in the Numeric Control library can be used.

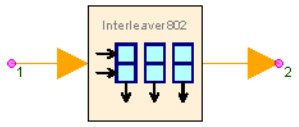
Interleaver802 Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Interleaver802 (algorithm)	IEEE 802 Interleaver

Interleaver802 (IEEE 802 Interleaver)



Description: IEEE 802 Interleaver
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Interleaver802 Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
s	Modular factor of interleaving	1		Integer	NO	[1:∞)
l	Divisor factor of interleaving	16		Integer	NO	[1:∞)
NCBPS	Number of coded bits per OFDM symbol	48		Integer	NO	[1:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	In	Input	int	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Out	Output	int	NO

Notes/Equations

1. The Interleaver802 model performs interleaving based on IEEE 802 standards. It performs the inverse of the Deinterleaver802 model.
2. This model reads N_{CBPS} samples from input In and writes N_{CBPS} samples to the output Out.
   - N_{CBPS} is the number of coded bits in a single OFDM symbol and is defined in the table below.
3. Interleaving is defined by a two-step permutation. The first permutation ensures that adjacent coded bits are mapped onto nonadjacent subcarriers. The second permutation ensures that adjacent coded bits are mapped alternately onto less and more significant bits of the constellation, thereby avoiding long runs of low reliability bits.
   In the following, k denotes the index of the coded bit before the first permutation; i denotes the index after the first and before the second permutation; j denotes the index after the second permutation, just prior to modulation mapping.
   The first permutation is defined by
   $i = (N_{CBPS}/l)(k \bmod l) + \text{floor}(k/l) \quad k = 0, 1, \ldots, N_{CBPS} - 1$
   The function floor (.) denotes the largest integer not exceeding the parameter.
   The second permutation is defined by
   $j = s \times \text{floor}(i/s) + (i + N_{CBPS} - \text{floor}(l \times i/N_{CBPS})) \bmod s \quad i = 0, 1, \ldots N_{CBPS} - 1$
   In the equations, *s is the modular factor and l is the divisor factor* ; these are variable parameters and their values depend on which standard the model is used for.
   If this model is used in IEEE 802.11 and HIPERLAN/2,
   $s = \max(N_{BPSC}/2, 1), l = 16;$
   where N_{BPSC} and N_{CBPS} are determined by data rates given in IEEE 802.11 and HIPERLAN/2 Rate-Dependent Values.
   If this model is used in IEEE 802.16,
   $s = N_{BPSC}/2, l = 12;$
   where N_{BPSC} and N_{CBPS} are determined by block sizes given in IEEE 802.16 Bit Interleaver Block Sizes (NCBPS /NBPSC).

IEEE 802.11 and HIPERLAN/2 Rate-Dependent Values

Data Rate (Mbps)	Modulation	Coding Rate (R)	Coded Bits per Subcarrier (NBPSC)	Coded Bits per OFDM Symbol (NCBPS)	Data Bits per OFDM Symbol (NDBPS)
6	BPSK	1/2	1	48	24
9	BPSK	3/4	1	48	36
12	QPSK	1/2	2	96	48
18	QPSK	3/4	2	96	72
24 (IEEE 802.11a)	16QAM	1/2	4	192	96
27 (HIPERLAN/2)	16QAM	9/16	4	192	108
36	16QAM	3/4	4	192	144
48 (IEEE 802.11a)	64QAM	2/3	6	288	192
54	64QAM	3/4	6	288	216

IEEE 802.16 Bit Interleaver Block Sizes (N_{CBPS} / N_{BPSC})

Modulation	16 Subchannels (Default)	8 Subchannels	4 Subchannels	2 Subchannels	1 Subchannel
QPSK	384/2	192/2	96/2	48/2	24/2
16QAM	768/4	384/4	192/4	96/4	48/4
64QAM	1152/6	576/6	288/6	144/6	72/6

See Also:

Deinterleaver802 (algorithm)

References

1. IEEE Standard 802.11a-1999, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: High-speed Physical Layer in the 5 GHz Band, 1999.
2. ETSI TS 101 475 v1.1.1, Broadband Radio Access Networks (BRAN); HIPERLAN Type 2; Physical (PHY) layer, April, 2000.
3. IEEE P802.16-REVd/D2-2003, *Part 16 Air Interface for Fixed Broadcast Wireless Access Systems*.

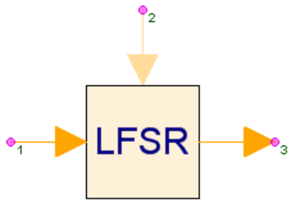
LFSR Part

Categories: *C++ Code Generation* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
LFSR (algorithm)	Linear Feedback Shift Register

LFSR



Description: Linear Feedback Shift Register
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *LFSR Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
FeedbackTaps	Feedback tap positions	[3, 10]		None	NO
InitialState	Initial state non-zero bit positions	[5, 9]		None	NO
UseInternalReset	Internal periodic reset mode: NO, YES	NO		Enumeration	NO
InternalResetPeriod	Number of outputs in a period	127		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	clock	int	NO
2	reset	int	YES

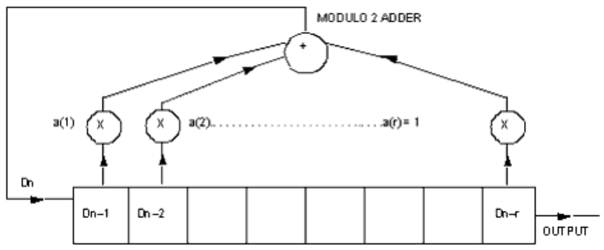
Output Ports

Port	Name	Signal Type	Optional
3	output	multiple int	NO

Notes/Equations

1. The linear feedback shift register (LFSR) part is configured to generate a PN sequence.
2. For every clock input, there is an LFSR output.
3. The output may be connected to a *bus* which can make all LFSR bits observable. A non-bus connection permit the standard output which is the rightmost bit in the LFSR Model. For a bus labeled B(a:b), B(a) is the standard output connection. If a<b, B(a+1) is the adjacent bit to the standard output and so on. This default connection order can be changed. Consult *Nets Connection Lines and Buses* (users). Note that any bus width specification greater than the LFSR number of bits generates an error.
4. New bit values are generated when the clock input is true (not 0) and the reset input is false (0), if the reset input is connected. Otherwise, the previous bit value is output.
5. When the reset input is true (not 0), the bits of InitialState is available for output.
6. If UseInternalReset is *YES*, then any reset input is ignored. The LFSR starts from InitialState every InternalResetPeriod samples.
7. The parameters FeedbackTaps and InitialState permit configuration for a LFSR with length up to 64 bits.
8. Both parameters can be specified either as a bit array or as an array of positions having bit value 1. The implied length of the bit array notation is the number of array elements. For the positional notation, the implied length is the largest position value.
9. The LFSR length r is defined by the implied length of FeedbackTaps. For example, if FeedbackTaps is specified as [7 3 2 1] in the positional notation or equivalently [1 1 1 0 0 0 1] in the bit array notation, then the LFSR length is 7 bits.
10. The diagram below illustrates the LFSR model.

LFSR Model



Constants a(1), a(2), ... , a(r) are binary feedback coefficients that are specified in FeedbackTaps. The length of the LFSR is r. The LFSR states are labeled as D(n − 1), D(n − 2), ... ,D(n − r). The LFSR is shifted to right, i.e. D(n − r) is replaced by D(n − r + 1) and so on. State D(n − r) is the standard output. The LFSR is initialized by shifting in bits at D(n − 1) from InitialState. The rightmost bit of InitialState is shifted out first. If the length of the InitialState parameter is i and i < r, then (r - i) additional zeros are shifted in. This results in InitialState shifted out beginning with the right most bit.

11. The following recurrence relation describes how the feedback value is calculated from the preceding r LFSR input bits.

$$D(n) = \left[\sum_{k=1}^{r} a(k)D(n-k)\right] \mathrm{mod2} \quad \text{for } n \geq 1$$

For example, if FeedbackTaps is [7, 3, 2, 1], then the feedback value is D(n) = (D(n − 7) + D(n − 3) + D(n − 2) + D(n − 1)) mod 2.

12. The following table provide a list of feedback coefficients for LFSR of length r which have the maximal code length property. Code length refers to the number of shifts of the LFSR before repeating the bit sequence.

Feedback Connections for Linear m-Sequences

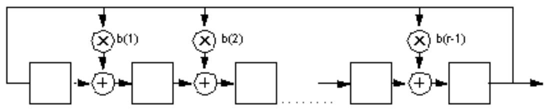
Number of Stages (r)	**Code Length**	**Maximal Feedback Taps**
2 [a]	3	[2, 1]
3 [a]	7	[3, 1]
4	15	[4, 1]
5 [a]	31	[5, 2] [5, 4, 3, 2] [5, 4, 2, 1]
6	63	[6, 1] [6, 5, 2, 1,] [6, 5, 3, 2,]
7 [a]	127	[7, 1] [7, 3] [7, 3, 2, 1,] [7, 4, 3, 2,] [7, 6, 4, 2] [7, 6, 3, 1] [7, 6, 5, 2] [7, 6, 5, 4, 2, 1] [7, 5, 4, 3, 2, 1]
8	255	[8, 4, 3, 2] [8, 6, 5, 3] [8, 6, 5, 2] [8, 5, 3, 1] [8, 6, 5, 1] [8, 7, 6, 1] [8, 7, 6, 5, 2, 1] [8, 6, 4, 3, 2, 1]
9	511	[9, 4] [9, 6, 4, 3] [9, 8, 5, 4] [9, 8, 4, 1] [9, 5, 3, 2] [9, 8, 6, 5] [9, 8, 7, 2] [9, 6, 5, 4, 2] [9, 7, 6, 4, 3, 1] [9, 8, 7, 6, 5, 3]
10	1023	[10, 3] [10, 8, 3, 2] [10, 4, 3, 1] [10, 8, 5, 1] [10, 8, 5, 4] [10, 9, 4, 1] [10, 8, 4, 3] [10, 5, 3, 2] [10, 5, 2, 1] [10, 9, 4, 2]
11	2047	[11, 2] [11, 8, 5, 2] [11, 7, 3, 2] [11, 5, 3, 5] [11, 10, 3, 2] [11, 6, 5, 1] [11, 5, 3, 1] [11, 9, 4, 1] [11, 8, 6, 2] [11, 9, 8, 3]
12	4095	[12, 6, 4, 1] [12, 9, 3, 2] [12, 11, 10, 5, 2, 1] [12, 11, 6, 4, 2, 1] [12, 11, 9, 7, 6, 5] [12, 11, 9, 5, 3, 1] [12, 11, 9, 8, 7, 4] [12, 11, 9, 7, 6, 5] [12, 9, 8, 3, 2, 1] [12, 10, 9, 8, 6, 2]
13 [a]	8191	[13, 4, 3, 1] [13, 10, 9, 7, 5, 4] [13, 11, 8, 7, 4, 1] [13, 12, 8, 7, 6, 5] [13, 9, 8, 7, 5, 1] [13, 12, 6, 5, 4, 3] [13, 12, 11, 9, 5, 3] [13, 12, 11, 5, 2, 1] [13, 12, 9, 8, 4, 2] [13, 8, 7, 4, 3, 2]
14	16,383	[14, 12, 2, 1] [14, 13, 4, 2] [14, 13, 11, 9] [14, 10, 6, 1] [14, 11, 6, 1] [14, 12, 11, 1] [14, 6, 4, 2] [14, 11, 9, 6, 5, 2] [14, 13, 6, 5, 3, 1] [14, 13, 12, 8, 4, 1] [14, 8, 7, 6, 4, 2] [14, 10, 6, 5, 4, 1] [14, 13, 12, 7, 6, 3] [14, 13, 11, 10, 8, 3]
15	32,767	[15, 1] [15, 4] [15, 13, 10, 9] [15, 13, 10, 1] [15, 14, 9, 2] [15, 9, 4, 1] [15, 12, 3, 1] [15, 10, 5, 4] [15, 10, 5, 4, 3, 2] [15, 11, 7, 6, 2, 1] [15, 7, 6, 3, 2, 1][15, 10, 9, 8, 5, 3] [15, 12, 5, 4, 3, 2] [15, 10, 9, 7, 5, 3] [15, 13, 12, 10] [15, 13, 10, 2] [15, 12, 9, 1] [15, 14, 12, 2] [15, 13, 9, 6] [15, 7, 4, 1] [15, 13, 7, 4]
16	65,535	[16, 12, 3, 1] [16, 12, 9, 6] [16, 9, 4, 3] [16, 12, 7, 2] [16, 10, 7, 6] [16, 15, 7, 2] [16, 9, 5, 2] [16, 13, 9, 6] [16, 15, 4, 2] [16, 15, 9, 4]
17 [a]	131,071	[17, 3] [17, 3, 2] [17, 7, 4, 3] [17, 16, 3, 1] [17, 12, 6, 3, 2, 1] [17, 8, 7, 6, 4, 3] [17, 11, 8, 6, 4, 2] [17, 9, 8, 6, 4, 1] [17, 16, 14, 10, 3, 2] [17, 12, 11, 8, 5, 2]

18	262,143	[18, 7] [18, 10, 7, 5] [18, 13, 11, 9, 8, 7, 6, 3] [18, 17, 16, 15, 10, 9, 8, 7] [18, 15, 12, 11, 9, 8, 7, 6]
19 [a]	524,287	[19, 5, 2, 1] [19, 13, 8, 5, 4, 3] [19, 12, 10, 9, 7, 3] [19, 17, 15, 14, 13, 12, 6, 1] [19, 17, 15, 14, 13, 9, 8, 4, 2, 1] [19, 16, 13, 11, 19, 9, 4, 1] [19, 9, 8, 7, 6, 3] [19, 16, 15, 13, 12, 9, 5, 4, 2, 1] [19, 18, 15, 14, 11, 10, 8, 5, 3, 2] [19, 18, 17, 16, 12, 7, 6, 5, 3, 1]
20	1, 048,575	[20, 3] [20, 9, 5, 3] [20, 19, 4, 3] [20, 11, 8, 6, 3, 2] [20, 17, 14, 10, 7, 4, 3, 2]
21	2,097,151	[21, 2] [21, 14, 7, 2] [21, 13, 5, 2] [21, 14, 7, 6, 3, 2] [21, 8, 7, 4, 3, 2] [21, 10, 6, 4, 3, 2] [21, 15, 10, 9, 5, 4, 3, 2] [21, 14, 12, 7, 6, 4, 3, 2] [21, 20, 19, 18, 5, 4, 3, 2]
22	4,194,303	[22,1] [22, 9, 5, 1] [22, 20, 18, 16,6, 4, 2, 1] [22, 19, 16, 13, 10, 7, 4, 1] [22, 17, 9, 7, 2, 1] [22, 17, 13, 12, 8, 7, 2, 1] [22, 14, 13, 12, 7, 3, 2, 1]
23	8,388,607	[23, 5] [23, 17, 11, 5] [23, 5, 4, 1] [23, 12, 5, 4] [23, 21, 7, 5] [23, 16, 13, 6, 5, 3] [23, 11, 10, 7, 6, 5] [23, 15, 10, 9, 7, 5, 4, 3] [23, 17, 11, 9, 8, 5, 4, 1] [23, 18, 16, 13, 11, 8, 5, 2]
24	16,777,215	[24, 7, 2] [24, 4, 3, 1] [24, 22, 20, 18, 16, 14, 11, 9, 8, 7, 5, 4] [24, 21, 19, 18, 17, 16, 15, 14, 13, 10, 9, 5, 4, 1]
25	33,554, 431	[25, 3] [25, 3, 2, 1] [25, 20, 5, 3] [25, 12, 5, 4] [25, 17, 10, 3, 2, 1] [25, 23, 21, 19, 9, 7, 5, 3] [25, 18, 12, 11, 6, 5, 4] [25, 20, 16, 11, 5, 3, 2, 1] [25, 12, 11, 8, 7, 6, 4, 3]
26	67,108,863	[26, 6, 2, 1] [26, 22, 21, 16, 12, 11, 10, 8, 5, 4, 3, 1]
27	134,217,727	[27, 5, 2, 1] [27, 18, 11, 10, 9, 5, 4, 3]
28	268,435,455	[28, 3] [28, 13, 11, 9, 5, 3] [28, 22, 11, 10, 4, 3] [28, 24, 20, 16, 12, 8, 4, 3, 2, 1]
29	536,870,911	[29, 2] [29, 20, 11, 2] [29, 13, 7, 2] [29, 21, 5, 2] [29, 26, 5, 2] [29, 19, 16, 6, 3, 2] [29, 18, 14, 6, 3, 2]
30	1,073,741,823	[30, 23, 2, 1] [30, 6, 4, 1] [30, 24, 20, 16, 14, 13, 11, 7, 2, 1]
31 [a]	2,147,483,647	[31, 29, 21, 17] [31, 28, 19, 15] [31, 3] [31, 3, 2, 1] [31, 13, 8, 3] [31, 21, 12, 3, 2, 1] [31, 20, 18, 7, 5, 3] [31, 30, 29, 25] [31, 28, 24, 10] [31, 20, 15, 5, 4, 3] [31, 16, 8, 4, 3, 2]
32	4,294,967,295	[32, 22, 2, 1] [32, 7, 5, 3, 2, 1] [32, 28, 19, 18, 16, 14, 11, 10, 9, 6, 5, 1]
33	8,589,934,591	[33, 13] [33, 22, 13, 11] [33, 26, 14, 10] [33, 6, 4, 1] [33, 22, 16, 13, 11, 8]
34	17,179,869,183	[34,27,2,1]
35	34,359,738,367	[35,33]
36	68,719,476,735	[36,25]
37	137,438,953,471	[37,5,4,3,2,1]
38	274,877,906,943	[38,6,5,1]
39	549,755,813,887	[39,35]
40	1,099,511,627,776	[40,38,21,19]
41	2,199,023,255,551	[41,38]
42	4,398,046,511,103	[42,41,20,19]
43	8,796,093,022,207	[43,42,38,37]
44	17,592,186,044,415	[44,43,18,17]
45	35,184,372,088,831	[45,44,42,41]
46	70,368,744,177,663	[46,45,26,25]
47	140,737,488,355,327	[47,42]
48	281,474,976,710,656	[48,47,21,20]
49	562,949,953,421,312	[49,40]
50	1,125,899,906,84,2623	[50,49,24,23]
51	2,251,799,813,685,248	[51,50,36,35]
52	4,503,599,627,370,496	[52,49]
53	9,007,199,254,740,991	[53,52,38,37]
54	18,014,398,509,481,983	[54,53,18,17]
55	36,028,797,018,963,967	[55,31]
56	72,057,594,037,927,935	[56,55,35,34]
57	144,115,188,075,855,871	[57,50]
58	288,230,376,151,711,743	[58,39]
59	576,460,752,303,423,488	[59,58,38,37]
60	1,152,921,504,606,846,975	[60,59]
61	2,305,843,009,213,693,951	[61, 5, 2, 1]
62	4,611,686,018,427,387,903	[62,61,6,5]
63	9,223,372,036,854,775,807	[63,62] [33, 13]
64	18,446,744,073,709,551,615	[64,63,61,60]

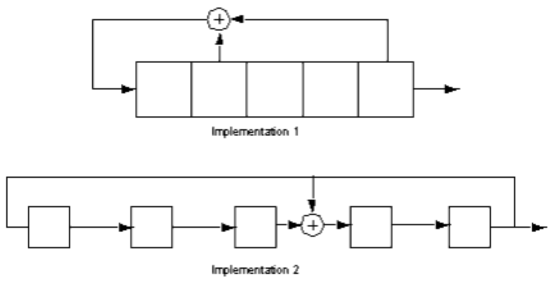
13. An alternative implementation of the LFSR is shown in Alternative Implementation of LFSR. In order to get the same output sequence from the two implementations the following relationships should hold between a(i) and b(i):

b(i) = a(r − i), i = 1, 2, ... , r − 1.
Implementation of 5-Stage LFSR illustrates implementation for a shift register of length 5 and FeedbackTaps = [2 5].
The sequence of the LFSR states in both implementations and the output (rightmost bit of the state) is shown in LFSR States.
Although the shift register in the two implementations does not go through the same sequence of states, the output sequence is the same for both. It is also worth noting that if the initial state is different from [1 0 0 0 0], the output sequences may not be identical but remain a shifted version of each other.

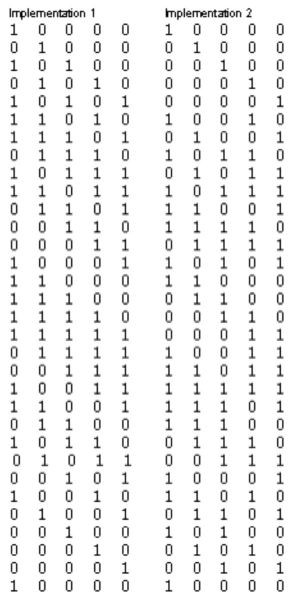
Alternative Implementation of LFSR



Implementation of 5-Stage LFSR



LFSR States



Implementation 1					Implementation 2				
1	0	0	0	0	1	0	0	0	0
0	1	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	1	0	0
0	1	0	1	0	0	0	0	1	0
1	0	1	0	1	0	0	0	0	1
1	1	0	1	0	1	0	0	1	0
1	1	1	0	1	0	1	0	0	1
0	1	1	1	0	1	0	1	1	0
1	0	1	1	1	0	1	0	1	1
1	1	0	1	1	1	0	1	1	1
0	1	1	0	1	1	1	0	0	1
0	0	1	1	0	1	1	1	1	0
0	0	0	1	1	0	1	1	1	1
1	0	0	0	1	1	0	1	0	1
1	1	0	0	0	1	1	0	0	0
1	1	1	0	0	0	1	1	0	0
1	1	1	1	0	0	0	1	1	0
1	1	1	1	1	0	0	0	1	1
0	1	1	1	1	1	0	0	1	1
0	0	1	1	1	1	1	0	1	1
1	0	0	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	0	1
0	1	1	0	0	1	1	1	0	0
1	0	1	1	0	0	1	1	1	0
0	1	0	1	1	0	0	1	1	1
0	0	1	0	1	1	0	0	0	1
1	0	0	1	0	1	1	0	1	0
0	1	0	0	1	0	1	1	0	1
0	0	1	0	0	1	0	1	0	0
0	0	0	1	0	0	1	0	1	0
0	0	0	0	1	0	0	1	0	1
1	0	0	0	0	1	0	0	0	0

See:
Bits
GoldCode (algorithm)

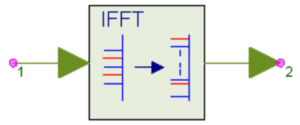
LoadIFFTBuff802 Part

Categories: *C++ Code Generation* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
LoadIFFTBuff802 (algorithm)	IEEE 802 IFFT Buffer Loader

LoadIFFTBuff802 (IEEE 802 IFFT Buffer Loader)



Description: IEEE 802 IFFT Buffer Loader
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *LoadIFFTBuff802 Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Carriers	Number of subcarriers per OFDM symbol	52		Integer	NO
DCCarrier	DC carrier: OFF, ON	OFF		Enumeration	NO
DCPilotValue	DC Pilot Value	1.33333		Complex number	YES
FullSubcarriers	Activate all sub-carriers: NO, YES	YES		Enumeration	NO
SubcarrierList	Subcarrier list	[-21, -7, 7, 21]		None	NO
Order	IFFT points as 2^Order	7		Integer	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	In	Transmitted signal before IFFT	complex	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Out	IFFT input signal, zero padded	complex	NO

Notes/Equations

1. This component is used to load transmission data into the IFFT buffer. Each firing, Carriers tokens are consumed and 2^{Order} tokens are generated. For example, if Carriers = 52, Order = 7, 52 tokens are consumed and 128 tokens are generated.
2. Data loading is performed as follows.
   Assume *x*(0), *x*(1), ... , *x*(*N*-1) are the inputs that generally represent active subcarriers defined by designers, where N = Carriers. *y*(0), *y*(1), ... , *y*(*M*-1) are the outputs, M = 2^{Order}.
   when N is even
   $$y(i) = x\left(\frac{N}{2}+i-1\right) \qquad i = 1, \ldots, \frac{N}{2}$$
   $$y(i) = 0 \qquad i = 0, \frac{N}{2}+1, \ldots, M-\frac{N}{2}-1$$
   $$y(i) = x\left(i-M+\frac{N}{2}\right) \qquad i = M-\frac{N}{2}, \ldots, M-1$$
   when N is odd
   $$y(i) = x\left(\frac{N-1}{2}+i-1\right) \qquad i = 1, \ldots, \frac{N+1}{2}$$
   $$y(i) = 0 \qquad i = 0, \frac{N+1}{2}+1, \ldots, M-\frac{N+1}{2}$$
   $$y(i) = x\left(i-M-\frac{N-1}{2}\right) \qquad i = M-\frac{N-1}{2}, \ldots, M-1$$
   For example, if Order = 4 and Carriers = 7, the input carriers are x(0), x(1), x(2), x(3),x(4),x(5),x(6), and the output carrier sequence would be:
   0 , x(3) , x(4) , x(5) , x(6) , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , x(0) , x(1) , x(2)
   which will be loaded into the IFFT model for the IFFT transformation.

3. DCCarrier and DCPilotValue specify whether DC carrier is used; if DCCarrier = ON, the DC carrier value is set by DCPilotValue.
   In the example provided in *note 2*, DCCarrier = OFF.
   While DCCarrier = ON and DCPilotValue = 4/3, the output carriers sequence would be:
   4/3, x(3), x(4), x(5), x(6), 0, 0, 0, 0, 0, 0, 0, 0, x(0), x(1), x(2)
   in which the first carrier is 4/3 instead of 0.
4. If FullSubcarriers = YES, all input carriers will be used. If FullSubcarriers = NO, some of the input carriers will be used; SubcarrierList specifies which input carriers will be used.
5. SubcarrierList (valid when FullSubcarriers = NO) specifies the positions of the input carriers to be used as active subcarriers (all subcarriers are 0 except those carriers specified).
   Assume *x*(0), *x*(1), ... , *x*(*N*-1) are the input signals that generally represent active subcarriers defined by designers, where N = Carriers. *y*(0), *y*(1), ... , *y*(*M*-1) are the output of the model $M = 2^{\text{Order}}$. The corresponding indices of *x*(0), *x*(1), ... , *x*(*N*-1) are {int(-Carriers/2), int(-Carriers/2) + 1, ... , -1, 1, ... , int(Carriers/2)-1, int(Carriers/2)}.
   The active subcarrier loading procedure is performed as follows: assume index is an element of {int(-Carriers/2), int(-Carriers/2) + 1, ... , -1, 1, ... , int(Carriers/2)-1, int(Carriers/2)}:
   when N is even
   $$y(index) = x\left(\frac{N}{2} + index - 1\right) \quad index > 0$$
   $$y(M + index) = x\left(index + \frac{N}{2}\right) \quad index < 0$$
   when N is odd
   $$y(index) = x\left(\frac{N-1}{2} + index - 1\right) \quad index > 0$$
   $$y(M + index) = x\left(index + \frac{N-1}{2}\right) \quad index < 0$$
   For example, SubcarrierList = {-2, -1, 2, 3}, and input carriers are x(0), x(1), x(2), x(3), x(4), x(5), x(6). Indices of the input carriers are -3, -2, -1, 1, 2, 3, 4. Elements in SubcarrierList must be integer and in (-Carriers/2, Carriers/2), in which Carriers is the number of carriers of input, here, it is 7 and index should be in [-3, 3]. In this case, the carrier with index is -2, -1, 2, 3 is used, these are x(1), x(2), x(4), x(5). The output subcarriers are then:
   4/3, 0, x(4), x(5), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, x(1), x(2).

References

1. IEEE Standard 802.11a-1999, "Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: High-speed Physical Layer in the 5 GHz Band," 1999.
2. ETSI TS 101 475 v1.1.1, "Broadband Radio Access Networks (BRAN); HIPERLAN Type 2; Physical (PHY) layer," April, 2000.
3. ARIB-JAPAN, Terrestrial Integrated Services Digital Broadcasting (ISDB-T); Specification of Channel Coding, Frame Structure and Modulation, Sept.1998.
4. ETSI, Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for digital terrestrial television. EN300 744 v1.2.1, European Telecommunication Standard, July 1999.
5. IEEE P802.15-03/268r1, "Multi-band OFDM Physical Layer Proposal for IEEE 802.15 Task Group 3a," September 2003.
6. IEEE P802.16-REVd/D2-2003, "Draft IEEE Standard for Local and metropolitan area networks Part 16: Air Interface for Fixed Broadband Wireless Access Systems," 2003.

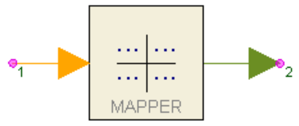
Mapper Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Mapper (algorithm)	Complex Symbol Mapper

Mapper (Complex Symbol Mapper)



Description: Complex Symbol Mapper
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Mapper Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
ModType	Modulation type: BPSK, QPSK, PSK8, PSK16, QAM16, QAM32, QAM64, QAM128, QAM256, User_Defined	QPSK		Enumeration	NO
MappingTable	Constellation table	[1, -1]		Complex array	NO
BitOrder	Bit order: LSB first, MSB first	LSB first		Enumeration	YES

Input Ports

Port	Name	Description	Signal Type	Optional
1	In	input bit sequence	boolean	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Out	output symbol sequence	complex	NO

Notes/Equations

1. Mapper groups consecutive bits as specified by the *BitOrder* parameter in the input to form a symbol value which is mapped to a complex valued constellation point that is output.
2. A constellation point is a pair of real values (I,Q) that is expressed on the output as I + jQ. Later in the modulation chain, I modulates the inphase part of the carrier, and Q modulates the quadrature part of the carrier over a symbol period.
3. Each modulation type has its constellation and symbol length. The symbol length, i.e. the number of input bits per symbol, is detailed in the following table.

   Modulation Type and Symbol Length

ModType	Symbol Length
BPSK	1
QPSK	2
PSK8	3
PSK16 or *QAM16*	4
QAM32	5
QAM64	6
QAM128	7
QAM256	8

   If ModType is *User_Defined* and the size of MappingTable is N, then the symbol length is $\log_2(N)$ bits.
4. For *QPSK*, *PSK8*, and *PSK16* the mapping from bits to symbols is using Gray encoding. For *QAM16*, *QAM32*, *QAM64*, *QAM128*, and *QAM256*, Gray encoding is used inside each quadrant.
5. For ModType *BPSK*, bit value 0 is mapped to 1 + j0 and bit value 1 is mapped to -1 + j0.
6. For ModType *QPSK*, the constellation map is illustrated in QPSK Constellation. For ModType *PSK8*, the constellation map is illustrated in 8PSK Constellation. For

ModType *PSK16*, the constellation map is illustrated in 16PSK Constellation.

7. The symbol mappings for ModType *QAM16*, *QAM32*, *QAM64*, *QAM128* and *QAM256* are described in the section 9 of [1], and their constellation maps are illustrated in figure 7-8 of [1].
8. QAM constellations need definition only for quadrant 1. The constellation points in quadrants 2, 3 and 4 are derived from quadrant 1 by selecting the quadrant 1 constellation value with the least significant bits of the input symbol and rotating that constellation value by the amount selected by the two most significant bits of the input symbol, b_i b_q, as specified in table Conversion of Constellation Points.

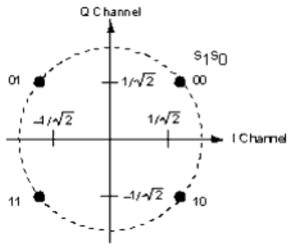
Conversion of Constellation Points

Quadrant	**Symbol Most Significant Bits (b_i b_q)**	**Rotation**
1	00	0
2	10	π/2
3	11	π
4	01	3π/2

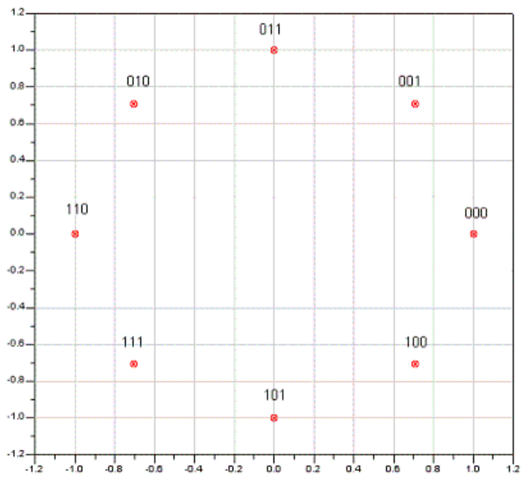
16QAM, 32QAM, 64QAM, 128QAM and 256QAM constellation maps are illustrated in 16 and 32QAM Constellation through 256QAM Constellation.

9. When ModType is specified to *User_Defined*, a custom constellation is defined with MappingTable. The input symbol is mapped directly to a constellation point as a 0 based index into MappingTable.

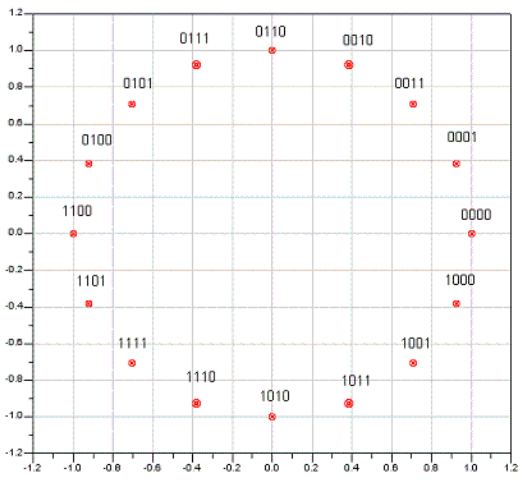
QPSK Constellation



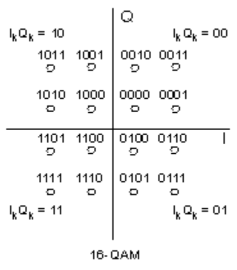
8PSK Constellation

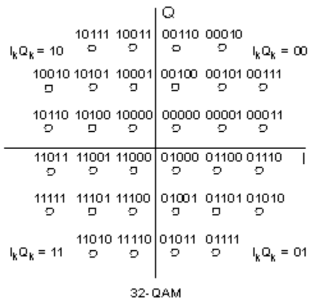


16PSK Constellation

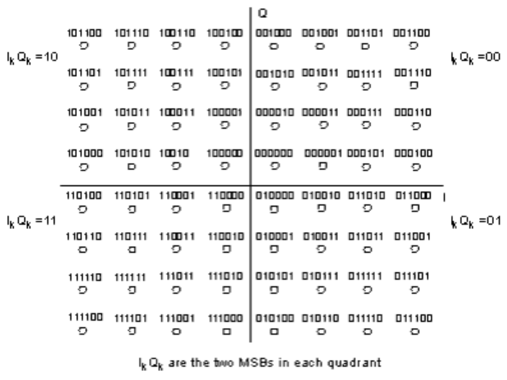


16 and 32QAM Constellation

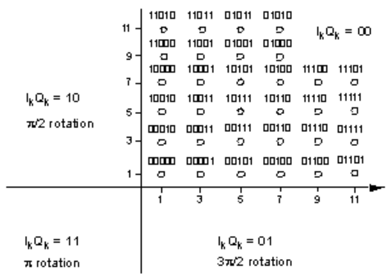




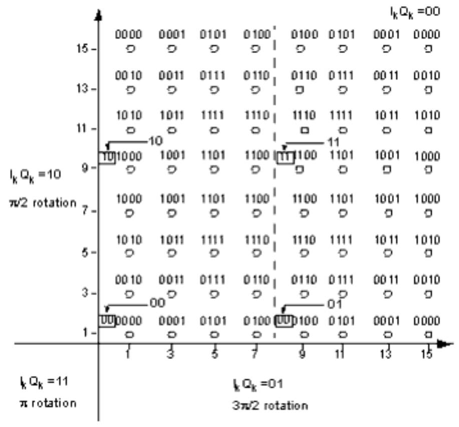
64QAM Constellation



128QAM Constellation



256QAM Constellation



See
Mapper_M (algorithm)
Demapper (algorithm)

References

1. EN 300 429, "Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for cable systems," V1.2.1, 1998-04.

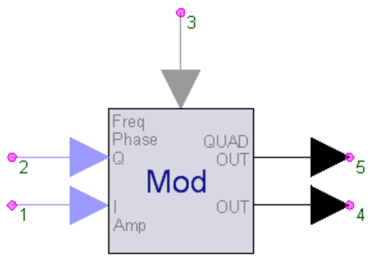
Modulator Part

Categories: *Analog/RF* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Modulator (algorithm)	Modulator

Modulator (Modulator)



Description: Modulator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Modulator Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
InputType	Input type: I/Q, Amp/Phase, Amp/Freq	I/Q		Enumeration	NO		
FCarrier	Carrier frequency (used if optional LO input not used)	0.2e6	Hz	Float	NO	[0:∞)	f_c
InitialPhase	Initial phase	0	deg	Float	NO	(-∞:∞)	θ
AmpSensitivity	Amplitude sensitivity	1		Float	NO	(-∞:∞)	S_a
PhaseSensitivity	Phase deviation sensitivity in degrees/Volt	90		Float	NO	(-∞:∞)	S_p
FreqSensitivity	Frequency deviation sensitivity in Hz/Volt	10000		Float	NO	(-∞:∞)	S_f
ConjugatedQuadrature	Negate quadrature output: NO, YES	NO		Enumeration	NO		
MirrorSignal	Mirror signal about carrier: NO, YES	NO		Enumeration	NO		
ShowIQ_Impairments	Show I and Q impairments: NO, YES	NO		Enumeration	NO		
GainImbalance	Gain imbalance in dB, Q channel relative to I channel	0.0		Float	NO	(-∞:∞)	G
PhaseImbalance	Phase imbalance, Q channel relative to I channel	0.0	deg	Float	NO	(-∞:∞)	φ
I_OriginOffset	I origin offset	0.0		Float	NO	(-∞:∞)	I_{off}
Q_OriginOffset	Q origin offset	0.0		Float	NO	(-∞:∞)	Q_{off}
IQ_Rotation	IQ rotation	0.0	deg	Float	NO	(-∞:∞)	R

Input Ports

Port	Name	Description	Signal Type	Optional
1	input1	input1	real	YES
2	input2	input2	real	YES
3	LO	complex envelope local oscillator signal	envelope	YES

Output Ports

Port	Name	Description	Signal Type	Optional
4	output	complex envelope vector output	envelope	NO
5	quad_output	complex envelope vector quadrature output	envelope	NO

Notes/Equations

1. The Modulator model implements a modulator that can perform amplitude, phase, frequency, or I/Q modulation.
2. This model reads 1 sample from the inputs and writes 1 sample to the outputs.
3. The *LO* input is optional. When not connected, an internal LO signal is used at the frequency *FCarrier*. When the external *LO* input is used, is must be a complex envelope signal with characterization frequency greater than zero. If use with a real signal is needed, then the *LO* input to this model can be preceded with an *EnvFcChange* (algorithm) model that will recharacterize a real signal to its representation at a specified frequency. The *LO* input can include power, phase, and noise variations in the LO signal. In the following, f_c is used to refer to the *FCarrier* parameter (when there is no *LO* input) or to the characterization frequency of the signal at the *LO* input.
4. When $f_c > 0$, the outputs are complex envelope signals with characterization frequency equal to f_c. The complex envelope is a complex value of the form I + j·Q.
5. When $f_c = 0$, the outputs are real baseband signals.
6. Basic operation for generating the modulated complex envelope signal (cx)
   - When *InputType* is set to *I/Q*, this model acts as an I/Q modulator. The I signal is applied at *input1* while the Q signal is applied at *input2*. The output signal is given by $cx = S_a \cdot (input1 + j \cdot input2)$

     Either input (but not both) can be left disconnected, in which case its value is assumed to be 0.
   - When *InputType* is set to *Amp/Phase*, this model acts as an amplitude and phase modulator. The amplitude modulating signal is applied at *input1*, and the phase modulating signal is applied at *input2*. The output signal is given by $cx = S_a \cdot input1 \cdot \exp(j \cdot input2 \cdot S_p \pi/180 + \theta)$

     Either input (but not both) can be left disconnected. If *input1* is left disconnected, then its value is assumed to be 1 and S_a is set to 1. In this case, this model acts as a pure phase modulator. If *input2* is left disconnected, then its value is assumed to be 0. In this case, this model acts as a pure amplitude modulator.
   - When *InputType* is set to *Amp/Freq*, this model acts as an amplitude and frequency modulator. The amplitude modulating signal is applied at *input1*, and the frequency modulating signal is applied at *input2*. The output signal is given by $cx = S_a \cdot input1 \cdot \exp(j \cdot 2\pi S_f \int input2 \cdot dt + \theta)$

     Either input (but not both) can be left disconnected. If *input1* is left disconnected, then its value is assumed to be 1 and S_a is set to 1. In this case, this model acts as a pure frequency modulator. If *input2* is left disconnected, then its value is assumed to be 0. In this case, this model acts as a pure amplitude modulator.
7. Effect of *MirrorSignal* on the modulated complex envelope signal (cx)
   - If *MirrorSignal* = YES, then let cx = conjugate(cx)
8. Effect of *GainImbalance*, *PhaseImbalance*, *I_OriginOffset*, *Q_OriginOffset*, and *IQ_Rotation* on the modulated complex envelope signal (cx)
   Assuming that cx is of the form I + j·Q, then the real signal that this complex envelope signal represents is I·cos(2πf_ct) − Q·sin(2πf_ct).
   - For *GainImbalance* and *PhaseImbalance*, let g = $10^{G/20}$. When these imbalances are applied to the modulator the real signal is I·cos(2πf_ct) − g·Q·sin(2πf_ct + φπ/180). Applying trigonometric formulas we can express this in the form I'·cos(2πf_ct) − Q'·sin(2πf_ct), where I'=I − g·Q·sin(φπ/180) and Q'=g·Q·cos(φπ/180) and so cx = I' + j·Q'.
   - For *IQ_Rotation*, cx = cx·(cos(*R*)+j·sin(*R*))
   - For *I_OriginOffset* and *Q_OriginOffset*, cx = cx + (I_{off} + j·Q_{off})
9. Obtaining the *output* and *quad_output* signals from the modulated complex envelope signal cx = Re{cx} + j·Im{cx}
   - When $f_c > 0$, *output* = cx and *quad_output* = −Im{cx} + j·Re{cx}
   - When $f_c = 0$, *output* = Re{cx} and *quad_output* = Im{cx}
10. See also: *Demodulator* (algorithm)

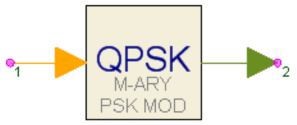
M_PSK Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
M_PSK (algorithm)	Complex PSK Symbol Mapper

M_PSK (Complex PSK Symbol Mapper)



Description: Complex PSK Symbol Mapper
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *M PSK Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
ModType	Modulation type: BPSK, QPSK, PSK8, PSK16, PSK32, PSK64, PSK128, PSK256, PSK512	QPSK		Enumeration	NO
BitOrder	Bit order: LSB first, MSB first	MSB first		Enumeration	YES

Input Ports

Port	Name	Description	Signal Type	Optional
1	In	Input bit sequence	int	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Out	Output complex symbol	complex	NO

Notes/Equations

1. M_PSK performs a M-ary phase shift key (PSK) modulation on the input bit stream, producing a Gray coded complex output signal. This component supports all popular M-ary PSK modulations in communication systems, including BPSK (2-BPSK), QPSK (4-PSK), 8-, 16-, 32-, 64-, 128-, 256-, and 512-PSK.
2. This is a multirate component. In general, if an M-ary PSK modulation is selected by using ModeType, the component consumes n = log2(M) bits from the input and produces one modulated complex output. Input bits are Gray encoded and mapped to an output constellation point as shown in *BPSK and QPSK Modulation Using Gray Encoding* to *32-PSK Modulation Using Gray Coding*. For example, if ModType = PSK8, the component consumes log2(8) = 3 bits from the input for Gray coded bits then maps these coded bits to a corresponding constellation point as shown in *8PSK Modulation Using Gray Coding*.
3. While there are many ways to encode and map sets of input bits into an M-point PSK constellation, Gray coding is always used for modulations to reduce error probabilities in communication systems. For M_PSK, a generic Labeling Expansion method proposed by E. Agrell [1] is used for Gray-encoding the input bits.
For specific mapping details, refer to *Mapper* (algorithm).
Figure: BPSK and QPSK Modulation Using Gray Encoding

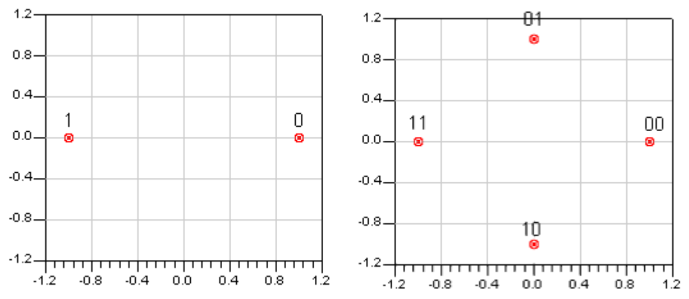


Figure: 8PSK Modulation Using Gray Coding

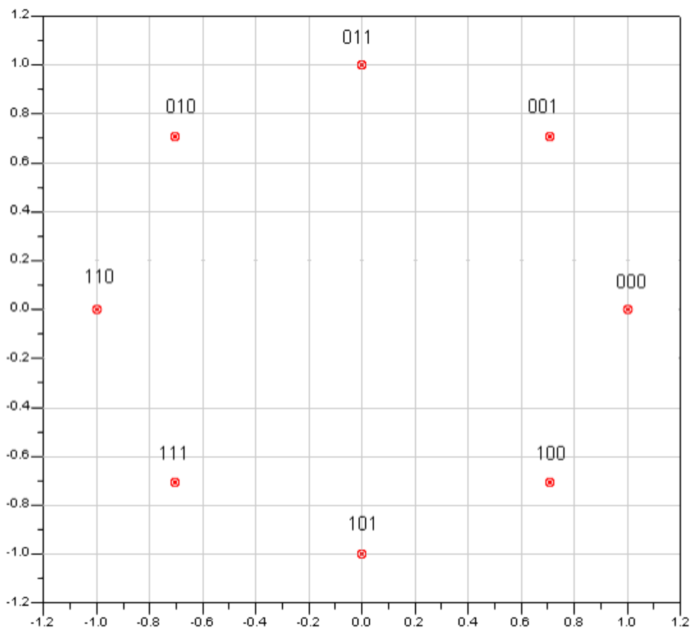


Figure: 16-PSK Modulation Using Gray Coding

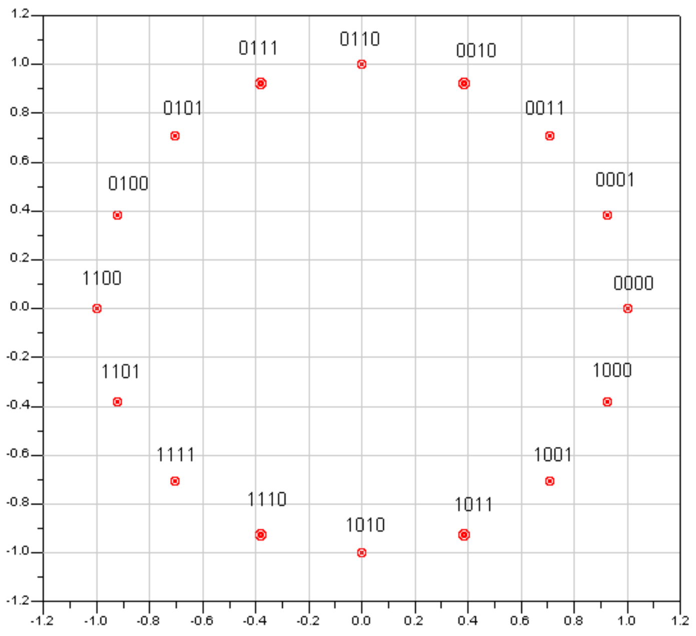
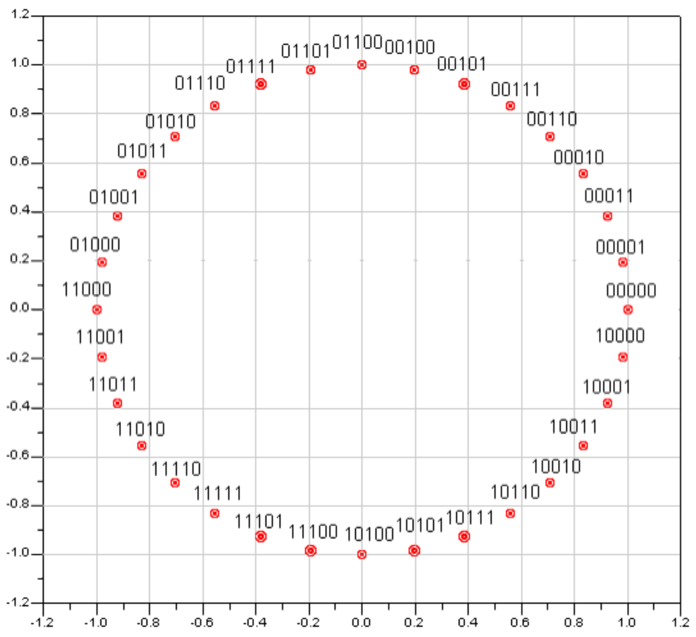


Figure: 32-PSK Modulation Using Gray Coding



References

1. E. Agrell, J.Lassing, E. G. Strüm, and T. Ottosson, "On the optimality of the binary reflected Gray code," *IEEE Transactions on Information Theory, vol. 50*, no. 12, pp. 3170-3182, Dec. 2004.
2. M. Jeruchim, P. Balaban and K. Shanmugan, *Simulation of Communication Systems*, Plenum Press, New York and London, 1992.

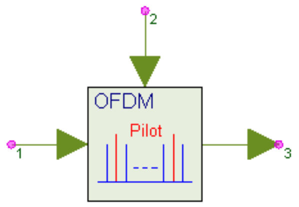
MuxOFDMSym802 Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
MuxOFDMSym802 (algorithm)	IEEE 802 OFDM Symbol Multiplexer

MuxOFDMSym802 (IEEE 802 OFDM Symbol Multiplexer)



Description: IEEE 802 OFDM Symbol Multiplexer
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *MuxOFDMSym802 Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Carriers	Number of subcarriers per OFDM symbol	52		Integer	NO	[1:8192]
DataCarriers	Number of data subcarriers per OFDM symbol	48		Integer	NO	[1:8192]
PilotPosition	Standard pilot positions	[-21, -7, 7, 21]		Integer array	NO	
PilotValue	Standard pilot values	[1, 1, 1, -1]		Complex array	NO	
GuardCarrierPosition	Guard carrier positions			Integer array	NO	
GuardCarrierValue	Guard carrier values			Complex array	NO	

Input Ports

Port	Name	Description	Signal Type	Optional
1	Data	data subcarriers input	complex	NO
2	Pilot	continual pilot value	complex	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	Out	OFDM symbol output	complex	NO

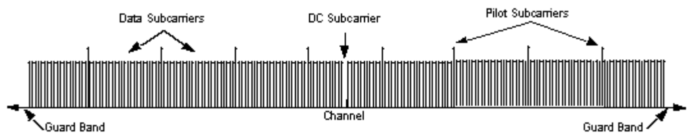
Notes/Equations

1. This component is used to multiplex data and pilot subcarriers into the OFDM symbol for IEEE 802 standards 802.11a, 802.11g, 802.15.3a, 802.16a, and 802.16d.

   Note
   OFDM symbols generally consist of continual pilots (CP) and scattered pilots (SP). Current IEEE 802 standards use CP only. Even though some DAB, DVB-T, and ISDB-T OFDM systems may use both CP and SP, MuxOFDMSym802 supports CP only.

2. The basic OFDM symbol structure is introduced in the frequency domain. The symbol (illustrated in OFDM Symbol) consists of subcarriers that determine the size of the FFT. There are several subcarrier types:
   - Data subcarriers for data transmission
   - Pilot subcarriers for estimations
   - Null subcarriers for no transmission, for guard bands and DC subcarrier.
     Guard bands in most OFDM systems (DVB-T, ISDB-T, 802.11a, 802.11g, 802.16a, and 802.16d) are inserted zeros.
     IEEE 802.15.3a has additional guard carriers defined between data subcarriers and guard bands. The guard subcarriers can be used for various purposes, including relaxing the specification on transmit and receive filters. The magnitude level of the guard tones is not specified, so reduced power levels for these subcarriers can be used. The all-zeros guard bands allow the signal to naturally decay and create the FFT *brick wall* shaping.

OFDM Symbol



This component multiplexes data and pilot subcarriers into one OFDM symbol according to the positions of data and pilot subcarriers defined in the standards. The null subcarriers (guard bands and DC subcarrier) are inserted into an OFDM symbol by the LoadIFFTBuff802 component. (Both MuxOFDMSym802 and LoadIFFTBuff802 components implement an OFDM symbol in the frequency domain.)

3. MuxOFDMSym802 parameter settings enable designers to generate a variety of OFDM symbol formats, in accordance with IEEE standards or not.
   Carriers specifies the number of active subcarriers (data subcarriers, pilot subcarriers and guard subcarriers) in one OFDM symbol.

   Note
   Carriers = DataCarriers _ PilotPosition _ GuardCarrierPosition.

   DataCarriers specifies the number of data subcarriers in one OFDM symbol.
   PilotPosition specifies continual pilot positions; PilotPosition is the number of pilot subcarriers in one OFDM symbol.
   PilotValue specifies values for continual pilot positions.
   GuardCarrierPosition specifies guard carriers positions (default = NULL); GuardCarrierPosition is the number of guard carrier subcarriers in one OFDM symbol.
   GuardCarrierValue specifies values for guard carrier positions (default = NULL).
4. Each firing, one Pilot token and DataCarriers tokens are consumed and Carriers tokens are output.
   The complex Data input signal is directly multiplexed into the OFDM symbol.
   The continual pilots are multiplexed into OFDM symbols as follows:
   p_k is the input in Pilot pin for kth OFDM symbol (or kth firing)

   $a_0, a_1, \dots, a_n$ are n+1 pilot values defined by PilotValue

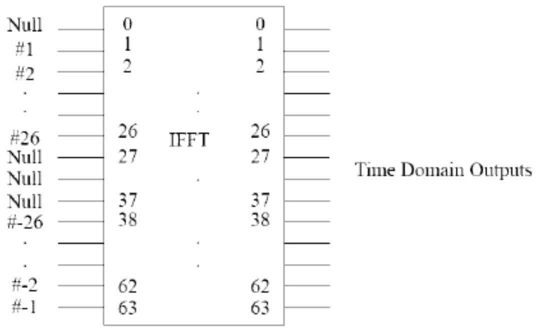
   The actual pilot values of kth OFDM symbol are $p_k \times a_0, p_k \times a_1, \dots, p_k \times a_n$.

   The continual pilot subcarrier values are multiplexed into the OFDM symbol according to PilotPosition.
   The guard carriers are multiplexed into the OFDM symbol like continual pilot as follows:
   $b_0, b_1, \dots, b_m$ are m+1 guard carriers values specified by GuardCarrierValue.

   The actual guard carrier values of kth OFDM symbol are $p_k \times b_0, p_k \times b_1, \dots, p_k \times b_m$.

   These guard carrier subcarriers values are multiplexed into the OFDM symbol according to GuardCarrierPosition.
5. The MuxOFDMSym802 output includes all active data, pilot, and guard carriers subcarriers indexed in the frequency domain:
   [-(*Carriers*)/2, -(*Carriers*)/2 + 1, ... , -1, 1, ... , (*Carriers* + 1)/2 -1, (*Carriers* + 1)/2]
   LoadIFFTBuff802 loads these output signals from MuxOFDMSym802 into the IFFT buffer and inserts zeros into the NULL and DC subcarriers. IFFT Input and Output (802.11a Specification) illustrates the 802.11a IFFT input and output. An OFDM symbol is input in the frequency domain after LoadIFFTBuff802; an OFDM symbol is output in the time domain after IFFT.

IFFT Input and Output (802.11a Specification)



References

1. IEEE Standard 802.11a-1999, "Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: High-speed Physical Layer in the 5 GHz Band," 1999.
2. ETSI TS 101 475 v1.1.1, "Broadband Radio Access Networks (BRAN); HIPERLAN Type 2; Physical (PHY) layer," April, 2000.
3. ARIB-JAPAN, Terrestrial Integrated Services Digital Broadcasting (ISDB-T); Specification of Channel Coding, Frame Structure and Modulation, Sept.1998.
4. ETSI, Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for digital terrestrial television. EN300 744 v1.2.1, European Telecommunication Standard, July 1999.
5. IEEE P802.15-03/268r1, "*Multi-band OFDM Physical Layer Proposal for IEEE 802.15 Task Group 3a*," September 2003.
6. IEEE P802.16-REVd/D2-2003, "*Draft IEEE Standard for Local and metropolitan area networks Part 16: Air Interface for Fixed Broadband Wireless Access Systems*," 2003.

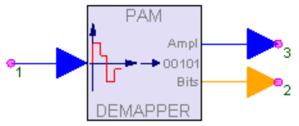
PAM_Demapper Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
PAM_Demapper (algorithm)	PAM Demapper/Slicer

PAM_Demapper (Pam Demapper)



Description: PAM Demapper/Slicer
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *PAM Demapper Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NumBits	Number of bits	4		Integer	NO	[1:∞)
BitOrder	Bit order: LSB first, MSB first	MSB first		Enumeration	YES	
LowLevel	Lowest level	-1		Float	YES	(-∞:∞)
HighLevel	Highest level	1		Float	YES	(LowLevel:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	Input	input signal	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Bits	output bit sequence	boolean	NO
3	Amplitude	closest PAM symbol	real	NO

Notes/Equations

1. The PAM_Demapper model implements a Pulse Amplitude Modulation demapper.
2. At every execution of the model, 1 sample is read from the input, *NumBits* samples are written to the *Bits* output, and 1 sample is written to the *Amplitude* output.
3. The demapping algorithm first finds which of the output levels of the corresponding PAM mapper is the closest to the input signal level, that is, find the *i* that minimizes | *Input* - L_i|, where

$$mean[n] = \frac{1}{(n+1)\cdot BlockSize}\cdot\sum_{i=0}^{(n+1)\cdot BlockSize-1} in[i]$$

$$variance[n] = \frac{1}{(n+1)\cdot BlockSize}\cdot\sum_{i=0}^{(n+1)\cdot BlockSize-1} in^2[i] - mean^2[n]$$

   Let *j* be that value of *i*. The integer *j* is converted to its binary representation in *NumBits* bits and these bit values are written to the *Bits* output (the *BitOrder* parameter defines whether the first bit written is the *LSB* or *MSB* of this binary representation). The level L_j is written to the *Amplitude* output.
4. See also: *PAM_Mapper* (algorithm)

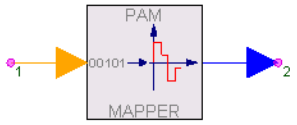
PAM_Mapper Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
PAM_Mapper (algorithm)	PAM Mapper

PAM_Mapper (PAM Mapper)



Description: PAM Mapper
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *PAM Mapper Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NumBits	Number of bits	4		Integer	NO	[1:∞)
BitOrder	Bit order: LSB first, MSB first	MSB first		Enumeration	YES	
LowLevel	Lowest level	-1		Float	YES	(-∞:∞)
HighLevel	Highest level	1		Float	YES	(LowLevel:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input bits	boolean	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output PAM symbol	real	NO

Notes/Equations

1. The PAM_Mapper model implements a Pulse Amplitude Modulation mapper.
2. At every execution of the model, *NumBits* samples (bits) are read from the input and mapped into a level between *LowLevel* and *HighLevel*, which is then written to the output.
3. The mapping algorithm first converts the *NumBits* bits read from the input (a 0 input value is considered to be a 0 bit; any other value is considered to be a 1 bit) into the equivalent integer representation (the *BitOrder* parameter defines whether the first bit read is to be considered as *LSB* or *MSB* in this bit to integer conversion). Let this integer be *i* (the range of values *i* can have is 0 to 2 ^{NumBits} − 1). Then the output level is given by

$$mean[n] = \frac{1}{(n+1)\cdot BlockSize}\cdot\sum_{i=0}^{(n+1)\cdot BlockSize-1} in[i]$$

$$variance[n] = \frac{1}{(n+1)\cdot BlockSize}\cdot\sum_{i=0}^{(n+1)\cdot BlockSize-1} in^2[i] - mean^2[n]$$

4. See also: *PAM_Demapper* (algorithm)

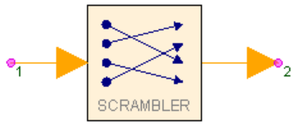
Scrambler Part

Categories: *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Scrambler (algorithm)	Bit Sequence Scrambler

Scrambler (Bit Sequence Scrambler)



Description: Bit Sequence Scrambler
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Scrambler Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Polynomial	Generator polynomial for the shift register - decimal, octal, or hex integer	147457		Integer	NO
ShiftReg	Initial state of the shift register - decimal, octal, or hex integer	1		Integer	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input bit sequence (zero or nonzero)	boolean	NO

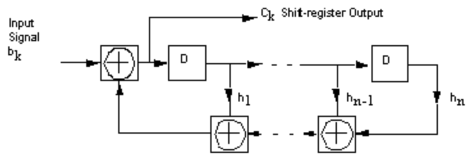
Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output bit sequence (zero or one)	boolean	NO

Notes/Equations

1. This component scrambles the input bit sequence using a feedback shift register, as shown in Feedback Shift Register. The taps of the feedback shift register are given by the Polynomial parameter, which should be a positive integer. The nth bit of this integer indicates whether the nth tap of the delay line is fed back. The low-order bit is called the 0th bit, and must be set. The next low-order bit indicates whether the output of the first delay should be fed back, and so on. The default Polynomial is an octal number defining the V.22bis scrambler.
2. In scramblers based on feedback shift registers, all the bits to be fed back are exclusive-ORed together (their parity is calculated), and the result is exclusive-ORed with the input bit. This result is produced at the output and shifted into the delay line. With proper choice of polynomial, the resulting output appears highly random even if the input is highly non-random (for example, all 0s or all 1s).

Feedback Shift Register



3. If the polynomial is a *primitive polynomial*, then the feedback shift register is a so-called *maximal length feedback shift register*. This means that with a constant input, the output will be sequence with period 2^N -1 where *N* is the order of the polynomial (the length of the shift register). This is the longest possible sequence. Moreover, within this period the sequence will appear to be white, in that a calculated autocorrelation will be very nearly an impulse. Therefore, the scrambler with a constant input can be very effectively used to generate a pseudo-random bit sequence.
   The maximal-length feedback shift register with constant input will pass through 2^N - 1 states before returning to a state it has been in before. This is one short of the 2^N

states that a register with *N* bits can take on. This one missing state, in fact, is a *lock-up* state, in that if the input is an appropriate constant, the scrambler will cease to produce random-looking output, and will output a constant. For example, if the input is all zeros, and the initial state of the scrambler is zero, then the outputs will be all zero, hardly random. This is easily avoided by initializing the scrambler to some non-0 state. That is why the default value for the ShiftReg parameter is set to 1.

4. The Polynomial parameter must be carefully chosen. It must represent a *primitive polynomial*, which is one that cannot be factored into two (nontrivial) polynomials with binary coefficients. For details, refer to [1].
5. The table below lists primitive polynomials (expressed as octal numbers so that these are easily translated into taps on shift register); these will result in maximal-length pseudo-random sequences if the input is constant and lockup is avoided.

Order	Polynomial	Order	Polynomial	Order	Polynomial
		11	04005	21	010000005
2	07	12	010123	22	020000003
3	013	13	020033	23	040000041
4	023	14	042103	24	0100000207
5	045	15	0100003	25	0200000011
6	0103	16	0210013	26	0400000107
7	0211	17	0400011	27	01000000047
8	0435	18	01000201	28	02000000011
9	01021	19	02000047	29	04000000005
10	02011	20	04000011	30	010040000007

The leading 0 in the polynomials indicates an octal number. Note also that reversing the order of the bits in any of these numbers will also result in a primitive polynomial. Therefore, the default value for the Polynomial parameter is 0440001 in octal, or "100 100 000 000 000 001" in binary. Reversing these bits we get "100 000 000 000 001 001" in binary, or 0400011 in octal. This latter number is listed in the table as the primitive polynomial of order 17. The order is the index of the highest-order non-0 bit in the polynomial, where the low-order bit has index 0.
Because the polynomial and the feedback shift register are both implemented using type *int*, the order of the polynomial is limited by the size of the *int* data type. For simplicity and portability, the polynomial is also not allowed to be interpreted as a negative integer, so the sign bit cannot be used. Therefore, if *int* is a 32-bit word, then the highest order polynomial allowed is 30 (recall that indexing for the order starts at 0, and we cannot use the sign bit). The primitive polynomials in the table are up to order 30 because of 32-bit integer machines.
Both the Polynomial and ShiftReg parameters can be set to a decimal, octal, or hex value. To enter an octal or hex value, prefix it with 0 or 0x, respectively. For example, in order to use the primitive polynomial of order 11, set Polynomial to 04005, 0x805, or 2053.
6. See also, *DeScrambler* (algorithm).

References

1. Lee and Messerschmitt, *Digital Communication*, Second Edition, Kluwer Academic Publishers, 1994, pp. 595-603.

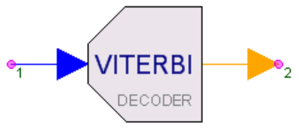
ViterbiDecoder Part

Categories: *C++ Code Generation* (algorithm), *Communications* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
ViterbiDecoder (algorithm)	Viterbi Decoder for Convolutional Code

ViterbiDecoder (Viterbi Decoder for Convolutional Code)



Description: Viterbi Decoder for Convolutional Code
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *ViterbiDecoder Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
CodingRate	Coding rate: rate_1_2, rate_1_3, rate_1_4, rate_1_5, rate_1_6, rate_1_7, rate_1_8	rate_1_2		Enumeration	NO
ConstraintLength	Constraint length	7		Integer	NO
Polynomial	Generator polynomial	[91, 121]		Integer array	NO
ZeroTail	Zero tail used to convert convolutional code to block code: NO, YES	NO		Enumeration	NO
BitSequenceLength	Length of bit squence not including tail bits	88		Integer	NO
MaxSurvivorLength	Maximum length of survivor in bits	35		Integer	NO
Polarity	Mapping mode from NRZ to logic signal: Negative to logic 1, Negative to logic 0	Negative to logic 1		Enumeration	NO
InitialState	Initial state of convolutional encoder: Zero state, Non-zero state	Zero state		Enumeration	NO
IgnoreNumber	Number of data points to be ignored	0		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	In	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	Out	boolean	NO

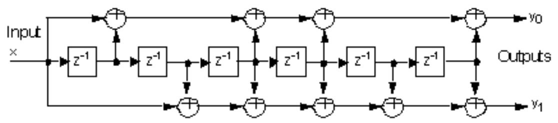
Notes/Equations

1. The ViterbiDecoder model is used for convolutionally decoding the input information sequence with a Viterbi algorithm.
2. If ZeroTail = NO, this model reads (1/R) samples from input *In* and writes 1 sample to output *Out*. If ZeroTail = YES, this model reads (N + K − 1)/R samples from input *In* and writes (N + K − 1) samples to output *Out*. R = 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, or 1/8 when CodingRate = rate_1_2, rate_1_3, rate_1_4, rate_1_5, rate_1_6, rate_1_7, or rate_1_8 respectively.
3. For example, the CDMA access channel CC(3, 1, 9) is defined with convolutional code rate R = 1/3 and K = 9. When *ZeroTail* = YES and N = 88, then this model reads 288 samples from input *In* and writes 96 samples to output *Out*.
4. Generally, there are two ways to implement convolutional code in communications system: code a semi-infinite bit sequence length where the initial encoder state could be zero- or non-zero with any final state; or, code block-by-block by appending zero tails after bit blocks so that the initial and the final encoder states are both zero. The ZeroTail parameter specifies this implementation; if ZeroTail = YES, then zero tails must be appended before input to this component.
5. ViterbiDecoder supports the 1/ *n* coding rate only. Convolutional codes with *k*/ *n* (*k* >1) are not supported by this component because: the coding and decoding will be more complex (this is also the reason why convolutional codes with a k/n (*k* >1) coding rate are seldom used in real communication systems); and, even convolutional codes with a *k*/ *n* (*k* >1) coding rate are used that are typically

implemented by puncturing the convolutional code with a 1/ *n* coding rate.

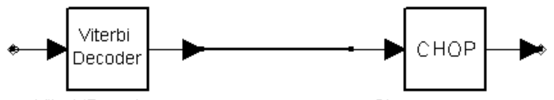
6. Polynomial is the convolutional code generator function. The generator matrix for a convolutional code is generally semi-infinite because the input sequence is semi-infinite. As an alternative to specifying the generator matrix, a functionally equivalent representation is used in which a set of n vectors is specified, one vector for each of the n modulo-2 adder. 1 in the ith position of the vector indicates that the corresponding stage in the shift register is connected to the modulo-2 adder; 0 in a given position indicates that no connection exists between that stage and the modulo-2 adder.
   For example, consider the binary convolutional encoder with constraint length K = 7, k = 1, and n = 2, illustrated in Convolutional Code CC(2,1,7). The connection for y0 is (1, 0, 1, 1, 0, 1, 1) from Outputs to Input, while the connection for y1 is (1, 1, 1, 1, 1, 0, 1). Generators for this code are conveniently given in octal form as (0133, 0175). So, when k=1, n generators (each of dimension K) are required to specify the encoder.

Convolutional Code CC(2,1,7)



7. ZeroTail is used to specify the encoder input sequence character. If ZeroTail = YES, the encoder input sequence is divided into blocks; block length is *N* . After each block, K−1 zeros are appended as tail bits. The total block length of the encoder is (*N* + K − 1), referring to Tail bits removal for ZeroTail = YES. In the decoder, known information can be used to obtain better performance.

Tail bits removal for ZeroTail = YES



ViterbiDecoder
V1
CodingRate=rate 1/2
ConstraintLength=ConstraintLength
Polynomial={0133, 0171}
ZeroTail=YES
BitSequenceLength=BitSequenceLength
MaxSurvivorLength=35
Polarity=Negative to logic 1

Chop
ExtraTailPSDU
nRead=BitSequenceLength+ConstraintLength-1
nWrite=BitSequenceLength
Offset=0
UsePastInputs=YES

8. BitSequenceLength (valid only when ZeroTail = YES) is used to specify the information bit length, which indicates the length of uncoded bits. This parameter can be set to the same value in the encoder and the decoder.
9. MaxSurvivorLength is the maximum length of the survivor that is stored in memory. The delay in decoding a long information sequence that has been convolutionally encoded is usually too long for most practical applications; moreover, memory required to store the entire length of surviving sequences is large and expensive. A solution for this is to modify the Viterbi algorithm in such a way that results in a fixed decoding delay without significantly affecting the optimal performance of the algorithm.
   The modification is to retain at any given time t only the most recent δ decoded informations bits in each surviving sequence. As each new information bit is received, a final decision is made on the bit received δ branches back in the trellis, by comparing the metrics in the surviving sequences and determining in favor of the bit in the sequence having the largest metric. If the δ chosen is sufficiently large, all surviving sequences will contain the identical decoded bit δ branches back in time. That is, with high probability, all surviving sequences at time t stem from the same one as t−. Experimental simulation has determined that a delay δ ≥ 5 *K* results in a negligible degradation in the performance relative to the optimum Viterbi algorithm.
10. Polarity is used to specify the mapping mode from bit (0, 1) to the NRZ signal level. Generally, bit 0 is mapped to level 1 and bit 1 is mapped level −1. An alternative is to map bit 0 to level −1 and bit 1 to level 1.
11. InitialState is used to specify the coded sequence character. If the initial state of encoder is zero-state, the known information can be used to obtain better performance. If the initial state is not known to be zero, InitialState must be set to a non-zero state.
12. IgnoreNumber is used to specify how much data will be ignored by this component. Delays in communications systems can be caused by devices or transmission. And, the delay may be inserted between the encoder and decoder in the form of meaningless data, so the information must be set in IgnoreNumber.
    - If ZeroTail = YES, the value of IgnoreNumber is *n* × (*N* + K − 1)/ *R* (*n* is an integer and *n* ≥ 0), and no extra delay will be introduced because it is assumed

the sequence is frame synchronized before input to ViterbiDecoder.

- If ZeroTail = NO, the delay is an integer number *n* ; this means the symbol synchronization is achieved before ViterbiDecoder. If *n* / *R* is also an integer, then the delay of output bit sequence will be *n* / *R* bits. Otherwise, the delay will be the minimum integer larger than *n* / *R*.
  Input sequence requirements are:
  If ZeroTail = YES
- The input sequence must be frame synchronized; that is, IgnoreNumber must be *n* × *N* / *R* (*n* is an integer and *n* ≥ 0) and the first valid data must be the first symbol of the first codeword in that frame.
- The input sequence must be encoded from blocks, each having K−1 zero tails so that the initial state and final state are all zero-state.
  If ZeroTail = NO
- The input sequence must be bit synchronized; that is, the first valid data must be the first symbol of a codeword.
- If InitialState is set to Zero state, the first valid symbol must be encoded with zero initial state.

13. The Viterbi algorithm is an optimal method of decoding convolutional codes. Optimal decoding decisions cannot be made on a symbol-by-symbol basis; instead, the entire received sequence must be compared with all possible transmitted sequences. The number of possible transmitted sequences increases exponentially with time, so an efficient method of comparing sequences is necessary.
The Viterbi algorithm is computationally efficient, but its complexity increases exponentially with the constraint length of the code. The Viterbi decoder measures how similar the received sequence is to a transmitted sequence by calculating a number called *path metric* (*path metric* of a sequence is calculated by adding numbers known as *symbol metric*, which is a measure of how close a received symbol is to each of the possible transmitted symbols). The transmitted sequence corresponding to the smallest path metric is declared to be the most likely sequence.
The Viterbi algorithm for a CC(n, k, K) code is described in the following paragraphs.
Branch Metric Calculation
The branch metric $m^{(a)}_j$, at the *J* th instant of the α path through the trellis is defined as the logarithm of the joint probability of the received n-bit symbol $r_j 1$, $r_j 2$ ... , r_{jn} conditioned on the estimated transmitted n-bit symbol $c_j 1^{(a)}$, $c_j 2^{(a)}$ ... , $c_{jn}^{(a)}$ for the α path. That is,

$$m^{(\alpha)}_j = \ln\left(\prod_{i=1}^{n} P(r_{ji}|c_{ji}{}^{(\alpha)})\right)$$

$$= \sum_{i=1}^{n} \ln P(r_{ji}|c_{ji}{}^{(\alpha)}).$$

If Rake receiver is regarded as a part of the channel, for the Viterbi decoder the channel can be considered to be an AWGN channel. Therefore,

$$m^{(\alpha)}_j = \sum_{i=1}^{n} r_{ji}c_{ji}$$

Path Metric Calculation
The path metric $M^{(a)}$ for the α path at the *J* th instant is the sum of the branch metrics belonging to the α path from the first instant to the *J* th instant. Therefore,

$$M^{(\alpha)} = \sum_{j=1}^{J} m^{(\alpha)}_j$$

Information Sequence Update
There are 2^k merging paths at each node in the trellis and the decoder selects from paths α1, α2, ... , α2k the one having the largest metric, namely:

$$max(M^{(\alpha_1)}, M^{(\alpha_2)}, \ldots, M^{(\alpha_{2^k})})$$

This path is known as the survivor.
Decoder Output
When the two survivors have been determined at the *J* th instant, the decoder outputs from memory the (*J-L*)th information symbol survivor with the largest metric.

14. **ViterbiDecoder Component Validation**
BER Measurements lists BER measurements for a rate 1/2 code (g_0 = 171, g_1 = 133) and a memoryless additive white Gaussian channel. Simulations were made with hard decision decoding (binary quantization) and soft decision decoding (no quantization). Simulation results are listed along with results published in QUALCOMM Technical Data Sheet Q0256; note that the published data and simulation results agree.
As can be seen from this table, there is a substantial BER performance improvement when this model is used with soft decision decoding. The difference between the two decoding processes is whether the input to this model is a binary signal (NRZ with -1 and 1 levels) or is an analog signal (the raw NRZ signal with noise and intersymbol

interference included). Where possible, use this model with an input analog signal with no binary decision made before this model. By providing the raw analog signal to the input of this model the model can then perform the soft decision decoding that results in superior BER performance.

Eb/No(dB)	**Hard Decision**		**Soft Decision**	
	Simulated BER	**QUALCOMM BER**	**Simulated BER**	**QUALCOMM BER (3 bits)**
3.0			3.62e-04	8.00e-04
3.5			7.56e-05	2.00e-04
4.0	5.01e-03	6.50e-03	1.11e-05	3.50e-05
4.5	1.79e-03	1.80e-03	2.12e-06	7.00e-06
5.0	5.71e-04	5.50e-04		
5.5	1.25e-04	9.00e-05		
6.0	2.81e-05	4.00e-05		

See Also:

ConvolutionalCoder (algorithm)

References

1. S. Lin and D. J. Costello, Jr., *Error Control Coding Fundamentals and Applications*, Prentice Hall, Englewood Cliffs NJ, 1983.
2. J. G. Proakis, Digital Communications (Third edition), Publishing House of Electronics Industry, Beijing, 1998.

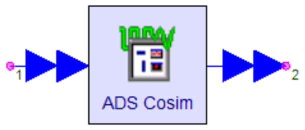
ADSCosimBlock Part

Categories: *Cosimulation* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model
ADSCosimBlock (algorithm)
ADSCosimBlockCx (algorithm)
ADSCosimBlockEnv (algorithm)

ADSCosimBlock



Description:
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *ADSCosimBlock Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
InputBlockSize	Array of numbers of input samples per port. The receiving end should set the same value.	[1]		None	NO
OutputBlockSize	Array of numbers of output samples per port. The sending end should set the same value.	[1]		None	NO
InputID	Identification for shared memory. The receiving end should set the same ID.	SystemVueToADS		Text	NO
OutputID	Identification for shared memory. The sending end should set the same ID.	ADSToSystemVue		Text	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	multiple real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	multiple real	NO

Notes/Equations

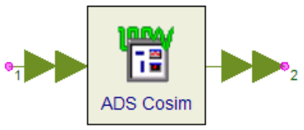
1. ADSCosimBlock performs SystemVue-ADS cosimulation by transferring double-type data samples between this model and the corresponding cosimulation blocks in ADS through shared memory.
2. This model consumes **InputBlockSize** numbers of samples from **input** ports and produces **OutputBlockSize** numbers of samples to **output** ports in every execution.
3. **InputBlockSize** is an array parameter specifying numbers of samples to be consumed for each of the **input** ports.
4. **OutputBlockSize** is an array parameter specifying numbers of samples to be produced to each of the **output** ports.
5. **InputID** specifies the shared memory identifier for transferring samples from the input port to the corresponding block *SVCosimSourceDbl* in ADS.
6. **outputID** specifies the shared memory identifier for transferring samples from the corresponding block *SVCosimSinkDbl* in ADS to the output port.
7. The input data samples are transferred to the corresponding ADS block *SVCosimSourceDbl* using shared memory. The **InputBlockSize** parameter in ADSCosimBlock and the **BlockSize** parameter in ADS *SVCosimSourceDbl* should be set to the same value. The **InputID** parameter in ADSCosimBlock and the **SVSenderID** parameter in ADS *SVCosimSourceDbl* should be set to the same identifier string.
8. The output data samples are transferred from the corresponding ADS block *SVCosimSinkDbl* using shared memory. The **OutputBlockSize** parameter in ADSCosimBlock and the **BlockSize** parameter in ADS *SVCosimSinkDbl* should be set to the same value. The **OutputID** parameter in ADSCosimBlock and the **SVReceiverID** parameter in ADS *SVCosimSinkDbl* should be set to the same identifier string.
9. The cosimulation library for ADS2009U1 is provided in \Examples\SV_ADS_Link\ADS2009U1\adsptolemy under SystemVue installation

directory. To load the cosimulation library in ADS, set ADSPTOLEMY_MODEL_PATH environment variable to "adsptolemy" directory (note that the path cannot contain any space). After starting ADS, the cosimulation blocks can be found in "SystemVue Cosimulation" category. For more details about custom models in ADS, please refer to http://edocs.soco.agilent.com/display/ads2009U1/Building+Signal+Processing+Models .

10. SystemVue-ADS cosimulation example workspaces can be found in \Examples\SV_ADS_Link under SystemVue installation directory.
11. When using this model for SystemVue-ADS cosimulation, both SystemVue and ADS workspaces should be opened and simulated on the same machine.
12. Use *ADSCosimBlockCx* (algorithm) for complex data type cosimulation. Use *ADSCosimBlockEnv* (algorithm) for timed envelope data type cosimulation.
13. Please refer to application note http://edocs.soco.agilent.com/display/eesofkc/SystemVue+ADS+link+bidirectional+cosim+3-in-1+example+LTE+HD+WPAN+ZigBee for details in SystemVue-ADS cosimulation examples. The application note also provides template projects for creating custom cosimulation models.

⚠ The current implementation does not provide time-out mechanism. If the communication cannot be established during simulation, you will have to terminate SystemVue process.

ADSCosimBlockCx



Description:
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *ADSCosimBlock Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
InputBlockSize	Array of numbers of input samples per port. The receiving end should set the same value.	[1]		None	NO
OutputBlockSize	Array of numbers of output samples per port. The sending end should set the same value.	[1]		None	NO
InputID	Identification for shared memory. The receiving end should set the same ID.	SystemVueToADS		Text	NO
OutputID	Identification for shared memory. The sending end should set the same ID.	ADSToSystemVue		Text	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	multiple complex	NO

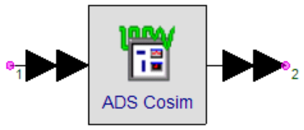
Output Ports

Port	Name	Signal Type	Optional
2	output	multiple complex	NO

Notes/Equations

1. ADSCosimBlockCx performs SystemVue-ADS cosimulation by transferring **complex**-type data samples between this model and the corresponding cosimulation blocks in ADS through shared memory.
2. This model consumes **InputBlockSize** numbers of samples from **input** ports and produces **OutputBlockSize** numbers of samples to **output** ports in every execution.
3. The corresponding cosimulation blocks in ADS are *SVCosimSourceCx* and *SVCosimSinkCx*, which are provided in \Examples\SV_ADS_Link\ADS2009U1\adsptolemy under SystemVue installation directory.
4. Please refer to *ADSCosimBlock* (algorithm) for detailed description and parameter specification.

ADSCosimBlockEnv



Description:
Domain: Timed

C++ Code Generation Support: NO
Associated Parts: *ADSCosimBlock Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
OutputFc	Array of characterization frequency of output envelope signal. The values should obtained from the sending co-simulation process.	0		None	NO
InputBlockSize	Array of numbers of input samples per port. The receiving end should set the same value.	[1]		None	NO
OutputBlockSize	Array of numbers of output samples per port. The sending end should set the same value.	[1]		None	NO
InputID	Identification for shared memory. The receiving end should set the same ID.	SystemVueToADS		Text	NO
OutputID	Identification for shared memory. The sending end should set the same ID.	ADSToSystemVue		Text	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	multiple envelope	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	multiple envelope	NO

Notes/Equations

1. ADSCosimBlockCx performs SystemVue-ADS cosimulation by transferring **envelope**-type data samples between this model and the corresponding cosimulation blocks in ADS through shared memory.
2. This model consumes **InputBlockSize** numbers of samples from **input** ports and produces **OutputBlockSize** numbers of samples to **output** ports in every execution.
3. The corresponding cosimulation blocks in ADS are *SVCosimSourceTimed* and *SVCosimSinkTimed*, which are provided in \Examples\SV_ADS_Link\ADS2009U1\adsptolemy under SystemVue installation directory.
4. **OutputFc** is an array parameter specifying characterization frequencies for each of the **output** ports (output envelope signal). **OutputFc** should be set according to the corresponding design in ADS.
5. Please refer to *ADSCosimBlock* (algorithm) for detailed description and parameter specification.

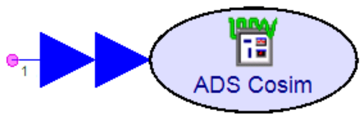
ADSCosimSink Part

Categories: *Cosimulation* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model
ADSCosimSink (algorithm)
ADSCosimSinkCx (algorithm)
ADSCosimSinkEnv (algorithm)

ADSCosimSink



Description:
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *ADSCosimSink Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
InputBlockSize	Array of numbers of input samples per port. The receiving end should set the same value.	[1]		None	NO
InputID	Identification for shared memory. The receiving end should set the same ID.	SystemVueToADS		Text	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	multiple real	NO

Notes/Equations

1. ADSCosimSink performs SystemVue-ADS cosimulation by transferring double-type data samples from this model to the corresponding cosimulation block in ADS through shared memory.
2. This model consumes **InputBlockSize** numbers of samples from **input** ports in every execution.
3. **InputBlockSize** is an array parameter specifying numbers of samples to be consumed for each of the **input** ports.
4. **InputID** specifies the shared memory identifier for transferring samples from the input port to the corresponding block *SVCosimSourceDbl* in ADS.
5. The input data samples are transferred to the corresponding ADS block *SVCosimSourceDbl* using shared memory. The **InputBlockSize** parameter in ADSCosimBlock and the **BlockSize** parameter in ADS *SVCosimSourceDbl* should be set to the same value. The **InputID** parameter in ADSCosimBlock and the **SVSenderID** parameter in ADS *SVCosimSourceDbl* should be set to the same identifier string.
6. The cosimulation library for ADS2009U1 is provided in \Examples\SV_ADS_Link\ADS2009U1\adsptolemy under SystemVue installation directory. To load the cosimulation library in ADS, set ADSPTOLEMY_MODEL_PATH environment variable to "adsptolemy" directory (note that the path cannot contain any space). After starting ADS, the cosimulation blocks can be found in "SystemVue Cosimulation" category. For more details about custom models in ADS, please refer to http://edocs.soco.agilent.com/display/ads2009U1/Building+Signal+Processing+Models .
7. SystemVue-ADS cosimulation example workspaces can be found in \Examples\SV_ADS_Link under SystemVue installation directory.
8. When using this model for SystemVue-ADS cosimulation, both SystemVue and ADS workspaces should be opened and simulated on the same machine.
9. Use *ADSCosimSinkCx* (algorithm) for complex data type cosimulation. Use *ADSCosimSinkEnv* (algorithm) for timed envelope data type cosimulation.
10. Please refer to application note http://edocs.soco.agilent.com/display/eesofkc/SystemVue+ADS+link+bidirectional+cosim+3-in-1+example+LTE+HD+WPAN+ZigBee for details in SystemVue-ADS cosimulation examples. The application note also provides template projects for creating custom cosimulation models.

⚠ The current implementation does not provide time-out mechanism. If the communication cannot be established during simulation, you will have to terminate SystemVue process.

ADSCosimSinkCx

Description:
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *ADSCosimSink Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
InputBlockSize	Array of numbers of input samples per port. The receiving end should set the same value.	[1]		None	NO
InputID	Identification for shared memory. The receiving end should set the same ID.	SystemVueToADS		Text	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	multiple complex	NO

Notes/Equations

1. ADSCosimSinkCx performs SystemVue-ADS cosimulation by transferring **complex**-type data samples from this model to the corresponding cosimulation block in ADS through shared memory.
2. This model consumes **InputBlockSize** numbers of samples from **input** ports in every execution.
3. The corresponding cosimulation block in ADS is *SVCosimSourceCx*, which is provided in \Examples\SV_ADS_Link\ADS2009U1\adsptolemy under SystemVue installation directory.
4. Please refer to *ADSCosimSink* (algorithm) for detailed description and parameter specification.

ADSCosimSinkEnv

Description:
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *ADSCosimSink Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
InputBlockSize	Array of numbers of input samples per port. The receiving end should set the same value.	[1]		None	NO
InputID	Identification for shared memory. The receiving end should set the same ID.	SystemVueToADS		Text	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	multiple envelope	NO

Notes/Equations

1. ADSCosimSinkEnv performs SystemVue-ADS cosimulation by transferring **envelope**-type data samples from this model to the corresponding cosimulation block in ADS through shared memory.
2. This model consumes **InputBlockSize** numbers of samples from **input** ports in every execution.
3. The corresponding cosimulation block in ADS is *SVCosimSourceTimed*, which is provided in \Examples\SV_ADS_Link\ADS2009U1\adsptolemy under SystemVue installation directory.
4. Please refer to *ADSCosimSink* (algorithm) for detailed description and parameter specification.

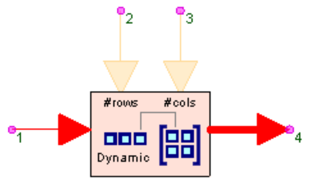
DynamicPack_M Part

Categories: *Dynamic* (algorithm), *Math Matrix* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
DynamicPack_M (algorithm)	Pack variable numbers of samples into matrices

DynamicPack_M



Description: Pack variable numbers of samples into matrices
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *DynamicPack M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Format	Format of data to be packed into matrix: ColumnMajor, RowMajor	ColumnMajor		Enumeration	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	Input samples	anytype	NO
2	numRows	Control signal for number of rows	int	YES
3	numColumns	Control signal for number of columns	int	YES

Output Ports

Port	Name	Description	Signal Type	Optional
4	output	Output matrix	anytype matrix	NO

Notes/Equations

1. DynamicPack_M packs variable numbers of input samples into output matrices based on *numRows* and *numColumns* control signals.
2. A complete execution consists of two modes: control mode and operation mode.
3. In control mode, DynamicPack_M reads one sample from numRows and one sample from numColumns, and the integer values specify *number-of-rows* and *number-of-columns* respectively for the matrix to be packed in the operation mode.
4. In operation mode, DynamicPack_M reads *number-of-rows* * *number-of-columns* number of samples from input, then packs them into *number-of-rows* by *number-of-columns* output matrix in either column major or row major according to the *Format* parameter.
5. If numRows is not connected, *number-of-rows* is default to be 1, which means a row vector. If numColumns is not connected, *number-of-columns* is default to be 1, which means a column vector. At least one of the control inputs (numRows, numColumns) must be connected.
6. The integer value for *number-of-rows* or *number-of-columns* can be 0 or negative, which results in an empty matrix.

See:
DynamicUnpack_M (algorithm)
Pack_M (algorithm)
Unpack_M (algorithm)

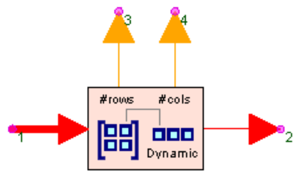
DynamicUnpack_M Part

Categories: *Dynamic* (algorithm), *Math Matrix* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
DynamicUnpack_M (algorithm)	Unpack variable-size matrices into samples

DynamicUnpack_M



Description: Unpack variable-size matrices into samples
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *DynamicUnpack M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Format	Format of data to be packed into matrix: ColumnMajor, RowMajor	ColumnMajor		Enumeration	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	Input matrix	anytype matrix	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	Output samples	anytype	NO
3	numRows	Number of rows	int	NO
4	numColumns	Number of columns	int	NO

Notes/Equations

1. DynamicUnpack_M unpacks variable-size input matrices into output samples.
2. A complete execution consists of two modes: control mode and operation mode.
3. In control mode, DynamicUnpack_M reads one matrix from input. Suppose the size of the matrix is *number-of-rows* by *number-of-columns*.
4. In operation mode, DynamicUnpack_M unpacks the matrix into *number-of-rows* * *number-of-columns* number of output samples in either column major or row major according to the *Format* parameter. It also writes *number-of-rows* integer value to numRows and writes *number-of-columns* integer value to numColumns.
5. If an input matrix is empty, DynamicUnpack_M will not generate any sample at output, but still write 0 to numRows and 0 to numColumns.

See:
DynamicPack_M (algorithm)
Pack_M (algorithm)
Unpack_M (algorithm)

Filters

The Filters library contains various digital (discrete-time) filters for signal processing. These filters are implemented as either digital FIR (finite impulse response) or digital IIR (infinite impulse response) filters.

For details, please refer *Common Filter Parameters* (algorithm)

Contents

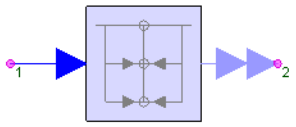
BiquadCascade Part

Categories: *C++ Code Generation* (algorithm), *Filters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
BiquadCascade (algorithm)	IIR Filter with Cascaded Biquad Sections

BiquadCascade (IIR Filter with Cascaded Biquad Sections)



Description: IIR Filter with Cascaded Biquad Sections
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *BiquadCascade Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Taps	Sets of six biquad coefficients.	[0.067455, 0.135, 0.067455, 1, -1.143, 0.4128]		Floating point array	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	input		real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	The outputs from each of the biquads in the cascade, starting with the output from last.	multiple real	NO

Notes/Equations

1. BiquadCascade is a cascade of 2-pole, 2-zero digital IIR filter (a biquad). This IIR filter has a Z-domain transfer function of

$$H(z) = \Pi \frac{Yi(z)}{Xi(z)} = \Pi \frac{N_{0i} + N_{1i} z^{-1} + N_{2i} z^{-2}}{D_{0i} + D_{1i} z^{-1} + D_{2i} z^{-2}}$$

2. For every input, one filtered value is output.
3. Each biquad section is defined by six coefficients in order: N $_{0,i}$ N $_{1,i}$ N $_{2,i}$ D $_{0,i}$ D $_{1,i}$ D $_{2,i}$.
4. The multi-output pin contains each of the outputs of the cascade, starting with the output from the last.
5. The default is a single biquad Butterworth filter with a frequency cutoff at 0.1 of the input rate.

See:
Biquad (algorithm)
IIR (algorithm)
IIR_Cx (algorithm)

References

1. A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall: Englewood Cliffs, NJ, 1989.

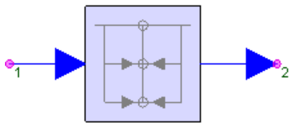
Biquad Part

Categories: *C++ Code Generation* (algorithm), *Filters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Biquad (algorithm)	Biquad IIR Filter

Biquad (Biquad IIR Filter)



Description: Biquad IIR Filter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Biquad Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
D1	First-order denominator coefficient	-1.143		Float	YES
D2	Second-order denominator coefficient	0.4128		Float	YES
N0	Zeroth-order numerator coefficient	0.067455		Float	YES
N1	First-order numerator coefficient	0.135		Float	YES
N2	Second-order numerator coefficient	0.067455		Float	YES

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

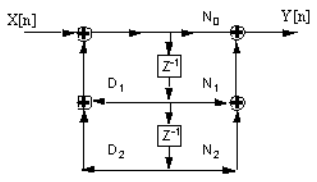
Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. Biquad is a 2-pole, 2-zero digital IIR filter (a biquad). This IIR filter has a Z-domain transfer function of
$$H(z) = \frac{Y(z)}{X(z)} = \frac{N_0 + N_1 z^{-1} + N_2 z^{-2}}{1 + D_1 z^{-1} + D_2 z^{-2}}$$
2. For every input, one filtered value is output.
3. *H(z)* results in the following second order difference equation.
$$y(n) = N_0 x(n) + N_1 x(n-1) + N_2 x(n-2) - D_1 y(n-1) - D_2 y(n-2)$$
where
y(n) is the output for sample *n*
x(n) is the input for sample *n*
4. *H(z)* is a linear time invariant system and can be rearranged to yield difference equation in direct form II as shown in Yield Difference Equation in Direct Form II. Indeed, it is the minimum number of delay elements required to implement a system with transfer function *H(z)*. An implementation with the minimum number of delay elements is also referred to as a canonic form implementation.
5. The default is a Butterworth filter with a frequency cutoff at 0.1 of the input rate.

Yield Difference Equation in Direct Form II



See:
BiquadCascade (algorithm)

IIR (algorithm)
IIR_Cx (algorithm)

References

1. A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall: Englewood Cliffs, NJ, 1989.

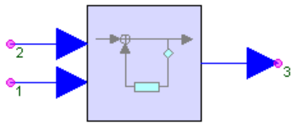
BlockAllPole Part

Categories: *Filters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
BlockAllPole (algorithm)	All-Pole Filter for Data Blocks

BlockAllPole (All-Pole Filter for Data Blocks)



Description: All-Pole Filter for Data Blocks
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *BlockAllPole Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
BlockSize	Number of inputs that use each coefficient set	128		Integer	NO	(0:∞)
Order	Number of new coefficients to read each time	16		Integer	NO	(0:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	signalIn		real	NO
2	coefs	Coefficients of the denominator polynomial	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	signalOut		real	NO

Notes/Equations

1. BlockAllPole implements an all-pole filter with coefficients that are periodically updated from the outside.
2. For a BlockSize number of inputs, an Order number of coefficients are read and a BlockSize number of filtered values are output.
3. The BlockSize parameter specifies how many input samples are processed using each set of coefficients.
4. The Order parameter tells how many coefficients are expected.
5. The transfer function of the filter is

$$H(z) = \frac{1}{1 - d_1 z^{-1} - d_2 z^{-2} - \ldots - d_M z^{-M}}$$

where d_i are the externally specified coefficients and M is the value of the Order parameter.

See:
IIR (algorithm)
IIR_Cx (algorithm)

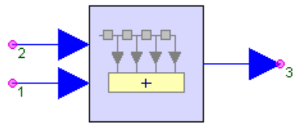
BlockFIR Part

Categories: *Filters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
BlockFIR (algorithm)	FIR Filter for Data Blocks

BlockFIR (FIR Filter for Data Blocks)



Description: FIR Filter for Data Blocks
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *BlockFIR Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
BlockSize	Number of inputs that use each coefficient set	128		Integer	NO	(0:∞)
Order	Number of new coefficients to read each time	16		Integer	NO	(0:∞)
Decimation	Decimation ratio	1		Integer	NO	(0:∞)
DecimationPhase	Decimation phase	0		Integer	NO	[0:_D_ecimation-1]
Interpolation	Interpolation ratio	1		Integer	NO	(0:∞)

Input Ports

Port	Name	Signal Type	Optional
1	signalIn	real	NO
2	coefs	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	signalOut	real	NO

{

Notes/Equations

1. BlockFIR implements an *FIR* (algorithm) filter with coefficients that are periodically updated from the outside.
2. For each set of Order number of coefficients, Decimation × BlockSize number of input samples is processed and a Interpolation × Blocksize number of filtered values are output.
3. BlockFIR efficiently implements sample rate changes. When the Decimation ratio is ≥1, the filter behaves as if it were followed by a *DownSample* (algorithm) part. When the Interpolation ratio is set, the filter behaves as if it were preceded by an *UpSample* (algorithm) part. The implementation is much more efficient, because a polyphase structure is used internally, thereby avoiding unnecessary use of memory and multiplications by 0. Arbitrary sample rate conversions by rational factors are accomplished this way.
4. The DecimationPhase parameter is equivalent to the Phase parameter of the *DownSample* (algorithm) part. When decimating, output samples are conceptually discarded. The polyphase structure does not calculate the discarded samples. To decimate by three, only one of every three outputs is selected. The DecimationPhase parameter determines which of these is selected. If DecimationPhase is 0 (default), the earliest outputs of the decimation block are decimated.
5. Avoid aliasing when designing a multi-rate filter. The filter sample rate is the product of the Interpolation parameter and the input sample rate or equivalently the product of the Decimation parameter and the output sample rate.

See:
FIR (algorithm)

References

1. F. J. Harris, "Multirate FIR Filters for Interpolating and Desampling," *Handbook of*

Digital Signal Processing, Academic Press, 1987.

Common Filter Parameters

This section introduces the definitions of some common filter parameters. Please note that a filter model contains only a subset of the parameters listed below, and may have specific parameters that are only described in its individual documentation. For the list of parameters for a particular filter model, please refer to *specific filter model documentation* (algorithm).

List of Common Filter Parameters

The common Filter parameters are listed below:

- **Loss:** The *Loss* parameter specifies the filter loss in *dB*.
- **FCenter:** The *FCenter* parameter specifies the passband center frequency in *Hz* for a bandpass filter or the stopband center frequency in *Hz* for a bandstop filter.
- **PassBandwidth:** The *PassBandwidth* parameter specifies the passband bandwidth in *Hz* for a bandpass filter or the difference between lower and upper passband edge frequencies in *Hz* for a bandstop filter.
- **PassFreq:** The *PassFreq* parameter specifies the passband edge frequency in *Hz* for highpass and lowpass filters.
- **PassAtten:** The *PassAtten* parameter specifies the attenuation in *dB* at the passband edge frequencies. Suppose a filter has -1dB passband edge magnitude, then *PassAtten* should be specified as 1.
- **PassRipple:** The *PassRipple* parameter specifies the passband ripple in *dB*. Suppose a filter requires -1dB passband ripple, then *PassRipple* should be specified as 1.
- **StopBandwidth:** The *StopBandwidth* parameter specifies the stopband bandwidth in *Hz* for a bandstop filter or the difference between lower and upper stopband edge frequencies in *Hz* for a bandpass filter.
- **StopFreq:** The *StopFreq* parameter specifies the stopband edge frequency in *Hz* for highpass and lowpass filters.
- **StopAtten:** The *StopAtten* parameter specifies the attenuation in *dB* at the stopband edge frequencies. Suppose a filter requires -30dB stopband edge magnitude, then *StopAtten* should be specified as 30.
- **StopRipple:** The *StopRipple* parameter specifies the stopband ripple in *dB*. Suppose a filter requires -30dB stopband ripple, then *StopRipple* should be specified as 30.
- **OrderType:** The *OrderType* parameter is used to select the filter order option: *User Defined* or *Auto*. When *OrderType* is *User Defined*, the *Order* parameter defines the order of the lowpass prototype analog filter (see *IIR Filter Design* (users)). On the other hand, when *OrderType* is *Auto*, the *Order* of the lowpass prototype analog filter is automatically computed.
- **Order:** For IIR filters, the *Order* parameter specifies the order of the lowpass prototype analog filter (see *IIR Filter Design* (users)). For FIR filters, the *Order* parameter specfies the filter order, which is equivalent to the number of filter coefficients (filter length) minus one.
- **MaximumOrder:** The *MaximumOrder* parameter specifies the maximum order in the iterative filter design process. In such process, the filter design stops at the *MaximumOrder*.
- **Transform:** The *Transform* parameter specifies S domain to Z domain transformation method for IIR filter design. The option can be either *Bilinear* Transform or *Impulse Invariance* (see *IIR Filter Design* (users)).
- **UnderSampledModel:** The *UnderSampledModel* parameter specifies the default behavior when the sampling rate is too small to represent the filter. The default behavior can be either *Model As Allpass* or *Error Out*. When the sampling rate is too small to represent the filter, *Model As Allpass* simply passes the input signal to the output without changing the signal. In contrast, the *Error Out* option issues error message and stops simulation. This parameter is only used for simulation. The *Filter Designer* (users) ignores this parameter.
- **Window:** The *Window* parameter specifies the window function applied for FIR filters. The possible window functions include *Rectangular*, *Bartlett*, *Hann*, *Hamming*, *Blackman*, *Flat Top*, *Generalized Cosine*, *Ready*, and *Kaiser* (see *Window Functions* (users)). An FIR filter model may only support a subset of window functions.
- **WindowParameter:** When *Window* is *Generalized Cosine*, *WindowParameter* specifies Generized Cosine coefficients [A B C D E ...] in as described in *Generized Cosine* (users). When *Window* is *Ready*, *WindowParameter* specifies γ

  as described in *Ready* (users)
- **Interpolation:** The *Interpolation* parameter specifies the interpolation factor for *Multirate Polyphase FIR Filter Implementation* (users). If the *Interpolation* factor is larger than 1, the input signal is up-sampled by the *Interpolation* factor with zero value insertion before the filter.
- **Decimation:** The *Decimation* parameter specifies the decimation factor for *Multirate Polyphase FIR Filter Implementation* (users). If the *Decimation* factor is larger than 1, the output signal is down-sampled by the *Decimation* factor with *Decimation Phase* after the filter.
- **DecimationPhase:** The *DecimationPhase* parameter specifies the decimation phase for *Multirate Polyphase FIR Filter Implementation* (users).
- **InterpolationScaling:** The *InterpolationScaling* parameter specifies whether the

output samples 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.

Filter Part

The SystemVue **Filter** part incorporates various filter models in terms of different frequency responses (e.g., lowpass, highpass, bandpass, bandstop, custom, etc.), IIR design methods (e.g., Bessel, Butterworth, ChebyshevI, ChebyshevII, Elliptic, Synchronously Tuned, S-domain poles-zeros, etc.), and FIR design methods (e.g., Parks-McClellan, Raised Cosine, Gaussian, Window, custom frequency response, etc.).

Filter Designer and Filter Part

SystemVue integrates the *Filter Designer (Filter Design Tool)* (users) with the *Filter Part* (algorithm). To launch the *Filter Designer* (users), place a **Filter part** on a schematic and **double click** it.
The Filter Design Tool, the Filter part, and most of the filter models associated with the Filter part are tightly integrated such that:

1. A specific *Response* and *Shape* (Design Method) selection in *Filter Designer* (users) corresponds to a specific filter model associated with the Filter part.

   If *Response* is Lowpass (LPF), Highpass (HPF), Bandpass (BPF), or Bandstop (BSF), the corresponding filter model is *Response_Shape*. For example, Lowpass Butterworth corresponds to *LPF_Butterworth* (algorithm), and Bandpass Parks-McClellan corresponds to *BPF_ParksMcClellan* (algorithm). Lowpass Half-Band Parks-McClellan is a specialized lowpass filter that maps to *HBLPF_ParksMcClellan* (algorithm).

   Under "Custom" *Response*, IIR "H(z) Coefficients (Z-Domain)" maps to *ZDomainIIR* (algorithm), IIR "H(s) Poles and Zeros (S-Domain)" maps to *SDomainSystem* (algorithm), FIR "Taps" maps to *ZDomainFIR* (algorithm), and FIR "Frequency Response" maps to *CustomFIR* (algorithm).
2. The parameters that appear in the *Filter Designer* (users) are the parameters of the chosen filter model.
3. After closing the *Filter Designer* (users), the last chosen filter model will be instantiated in the Filter part with the proper symbol.
4. All the parameter values of the instantiated filter model remain the same as they were specified in the *Filter Designer* (users).

For the filter models integrated with the Filter Designer, "double-click" the part opens the Filter Designer. To open the normal *Part Properties* (users) window, **right-click** on the part and select "Properties...".

Multirate Polyphase Implementation for FIR Filter Models

Most of the SystemVue FIR filter models support multirate capabilities (i.e., interpolation, decimation, and decimation phase) with efficient polyphase implementation. Please refer to *Multirate Polyphase FIR Filter Implementation* (users) for details.

Designing FIR Coefficients for Fixed-Point FIR Model

To design an FIR filter in *Filter Designer* (users) and set the coefficients to the fixed-point FIR model *FIR_Fxp* (hardware), this process can be done as follows:

1. Design an FIR filter in *Filter Designer* (users).
2. Switch to the "Coefficients" tab, click "Convert to Z-Domain Digital Model" button to convert the designed coefficients into *ZDomainFIR* (algorithm), and close Filter Designer Window.
3. Right click on the Filter part to open the Part Properties window, then switch the model from *ZDomainFIR* (algorithm) to *FIR_Fxp* (hardware). The pre-designed coefficients will remain the same in *FIR_Fxp* (hardware).

Filtering Envelope Signal

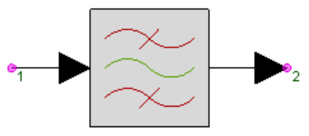
Most filter models associated with the Filter part support *Envelope Signal* (sim) (black pin). Please refer to *Filtering Envelope Signal* (sim) for technical details about how SystemVue filter models filter real and analytic signal.

Associated Models

The following table lists the models that are available/associated with this part. To view detailed information on a model, please select the appropriate link from the table. This information will include, but is not limited to: description, parameters, inputs, outputs, notes and equations.

Model	**Description**
BPF Bessel (algorithm)	Bandpass Bessel Filter
BPF Butterworth (algorithm)	Bandpass Butterworth Filter
BPF ChebyshevI (algorithm)	Bandpass Chebyshev I Filter
BPF ChebyshevII (algorithm)	Bandpass Chebyshev II Filter
BPF Elliptic (algorithm)	Bandpass Elliptic Filter
BPF Gaussian (algorithm)	Gaussian bandpass filter
BPF ParksMcClellan (algorithm)	Bandpass linear phase FIR filter based on Parks-McClellan design method
BPF RaisedCosine (algorithm)	Raised cosine bandpass filter
BPF SynchronouslyTuned (algorithm)	Bandpass SynchronouslyTuned Filter
BPF Window (algorithm)	Bandpass window-based linear phase FIR filter
BSF Bessel (algorithm)	Bandstop Bessel Filter
BSF Butterworth (algorithm)	Bandstop Butterworth Filter
BSF ChebyshevI (algorithm)	Bandstop Chebyshev I Filter
BSF ChebyshevII (algorithm)	Bandstop Chebyshev II Filter
BSF Elliptic (algorithm)	Bandstop Elliptic Filter
BSF ParksMcClellan (algorithm)	Bandstop linear phase FIR filter based on Parks-McClellan design method
BSF SynchronouslyTuned (algorithm)	Bandstop SynchronouslyTuned Filter
BSF Window (algorithm)	Bandstop window-based linear phase FIR filter
CustomFIR (algorithm)	Custom FIR filter
HBLPF ParksMcClellan (algorithm)	Half-Band Lowpass linear phase FIR filter based on Parks-McClellan design method
HPF Bessel (algorithm)	Highpass Bessel Filter
HPF Butterworth (algorithm)	Highpass Butterworth Filter
HPF ChebyshevI (algorithm)	Highpass Chebyshev I Filter
HPF ChebyshevII (algorithm)	Highpass Chebyshev II Filter
HPF Elliptic (algorithm)	Highpass Elliptic Filter
HPF ParksMcClellan (algorithm)	Highpass linear phase FIR filter based on Parks-McClellan design method
HPF SynchronouslyTuned (algorithm)	Highpass Synchronously Tuned Filter
HPF Window (algorithm)	Highpass window-based linear phase FIR filter
LPF Bessel (algorithm)	Lowpass Bessel Filter
LPF Butterworth (algorithm)	Lowpass Butterworth Filter
LPF ChebyshevI (algorithm)	Lowpass Chebyshev I Filter
LPF ChebyshevII (algorithm)	Lowpass Chebyshev II Filter
LPF Edge (algorithm)	EDGE pulse shaping lowpass filter
LPF Elliptic (algorithm)	Lowpass Elliptic Filter
LPF Gaussian (algorithm)	Gaussian lowpass filter
LPF ParksMcClellan (algorithm)	Lowpass linear phase FIR filter based on Parks-McClellan design method
LPF RaisedCosine (algorithm)	Raised cosine lowpass filter
LPF SynchronouslyTuned (algorithm)	Lowpass Synchronously Tuned Filter
LPF Window (algorithm)	Lowpass window-based linear phase FIR filter
SDomainSystem (algorithm)	S domain system
ZDomainFIR (algorithm)	Z domain FIR filter
ZDomainIIR (algorithm)	Z domain IIR filter
FIR_Fxp (hardware)	Fixed Point Finite Impulse Response Filter

BPF_Bessel (Bandpass Bessel Filter)



Description: Bandpass Bessel Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	50e3	Hz	Float	NO	(0:FCenter*2)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth (required only when OrderType = Auto)	125e3	Hz	Float	NO	(PassBandwidth:FCenter*2)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	20		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]
Transform	S to Z domain transformation method: Bilinear, Impulse Invariance	Bilinear		Enumeration	NO	
UnderSampledModel	Default behavior when sampling rate is too small to represent the filter. This parameter is only used for simulation not for filter design tool: Model As Allpass, Error Out	Model As Allpass		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

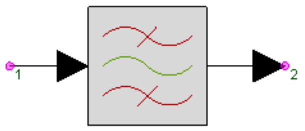
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BPF_Bessel implements a Bandpass Bessel IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

BPF_Butterworth (Bandpass Butterworth Filter)



Description: Bandpass Butterworth Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	50e3	Hz	Float	NO	(0:FCenter*2)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth (required only when OrderType = Auto)	100e3	Hz	Float	NO	(PassBandwidth:FCenter*2)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]
Transform	S to Z domain transformation method: Bilinear, Impulse Invariance	Bilinear		Enumeration	NO	
UnderSampledModel	Default behavior when sampling rate is too small to represent the filter. This parameter is only used for simulation not for filter design tool: Model As Allpass, Error Out	Model As Allpass		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

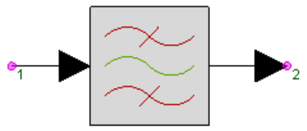
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BPF_Butterworth implements Bandpass Butterworth IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

BPF_ChebyshevI (Bandpass ChebyshevI Filter)



Description: Bandpass Chebyshev I Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	50e3	Hz	Float	NO	(0:FCenter*2)
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth (required only when OrderType = Auto)	100e3	Hz	Float	NO	(PassBandwidth:FCenter*2)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]
Transform	S to Z domain transformation method: Bilinear, Impulse Invariance	Bilinear		Enumeration	NO	
UnderSampledModel	Default behavior when sampling rate is too small to represent the filter. This parameter is only used for simulation not for filter design tool: Model As Allpass, Error Out	Model As Allpass		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

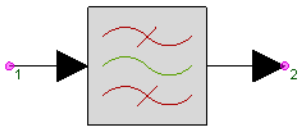
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BPF_ChebyshevI implements a Bandpass Chebyshev I IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

BPF_ChebyshevII (Bandpass ChebyshevII Filter)



Description: Bandpass Chebyshev II Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth (required only when OrderType = Auto)	50e3	Hz	Float	NO	(0:FCenter*2)
PassAtten	Passband edge attenuation in dB (required only when OrderType = Auto)	3		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth	100e3	Hz	Float	NO	(PassBandwidth:FCenter*2)
StopRipple	Stopband ripple in dB	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]
UnderSampledModel	Default behavior when sampling rate is too small to represent the filter. This parameter is only used for simulation not for filter design tool: Model As Allpass, Error Out	Model As Allpass		Enumeration	NO	

Input Ports

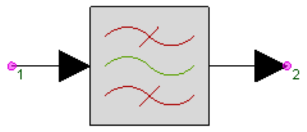
Port	Name	Signal Type	Optional
1	input	envelope	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

1. BPF_ChebyshevII implements a Bandpass Chebyshev II IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

BPF_Elliptic (Bandpass Elliptic Filter)



Description: Bandpass Elliptic Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	50e3	Hz	Float	NO	(0:FCenter*2)
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth	100e3	Hz	Float	NO	(PassBandwidth:FCenter*2)
StopRipple	Stopband ripple in dB (required only when OrderType = Auto)	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]
UnderSampledModel	Default behavior when sampling rate is too small to represent the filter. This parameter is only used for simulation not for filter design tool: Model As Allpass, Error Out	Model As Allpass		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

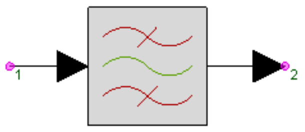
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BPF_Elliptic implements a Bandpass Elliptic IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

BPF_Gaussian (Bandpass Gaussian Filter)



Description: Gaussian bandpass filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	50e3	Hz	Float	NO	(0:FCenter/2)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
LengthOption	Filter length option (Auto: 1.2 / 3db frequency): Auto, Number of Taps, Impulse Time Duration	Auto		Enumeration	NO	
Length	Filter length (based on LengthOption, specify number of taps, or impulse time duration; delay = length/2)	0		Float	NO	[0:∞)
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

Output Ports

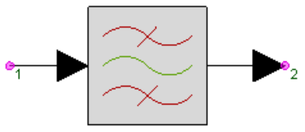
Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BPF_Gaussian implements a Bandpass Gaussian FIR filter.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *FIR Filter Design* (users) for the technical information about the design process.
5. The delay introduced by the filter is approximately equal to:

$$Delay = \frac{0.6}{\sqrt{3.0/(PassAtten)} \times PassFreq}$$

BPF_ParksMcClellan (Bandpass ParksMcClellan Filter)



Description: Bandpass linear phase FIR filter based on Parks-McClellan design method
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	50e3	Hz	Float	NO	(0:FCenter*2)
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth	100e3	Hz	Float	NO	(PassBandwidth:FCenter*2)
StopRipple	Stopband ripple in dB	30		Float	NO	[3.0103:∞]
MaximumOrder	Maximum filter order for Parks-McClellan filter design	300		Integer	NO	
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

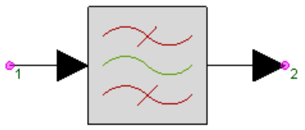
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BPF_ParksMcClellan implements a linear-phase Bandpass FIR filter using the Parks-McClellan design method.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *FIR Filter Design* (users) for the technical information about the design process.

BPF_RaisedCosine (Bandpass Raised Cosine Filter)



Description: Raised cosine bandpass filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
SymbolRate	Symbol rate (passband bandwidth = SymbolRate)	100e3	Hz	Float	NO	(0:FCenter/2)
RollOff	Roll-off factor (between 0 and 1)	0.5		Float	NO	[0:1]
SquareRoot	Square root option: NO, YES	NO		Enumeration	NO	
PulseEqualization	Pulse equalization option: NO, YES	NO		Enumeration	NO	
LengthOption	Filter length option (Auto: 8 * symbol period): Auto, Number of Taps, Number of Symbols, Impulse Time Duration	Auto		Enumeration	NO	
Length	Filter length (length = 2*delay) (based on LengthOption, specify number of taps, number of symbols, or impulse time duration)	0		Float	NO	[0:∞)
Window	Window function: Rectangular, Bartlett, Hann, Hamming, Blackman, Flat Top	Rectangular		Enumeration	NO	
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

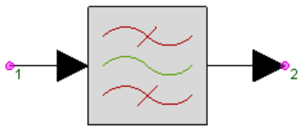
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BPF_RaisedCosine implements a Bandpass Raised Cosine FIR filter.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *FIR Filter Design* (users) for the technical information about the design process.

BPF_SynchronouslyTuned (Bandpass SynchronouslyTuned Filter)



Description: Bandpass SynchronouslyTuned Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	**Description**	**Default**	**Units**	**Type**	**Runtime Tunable**	**Range**
Loss	Magnitude loss in dB	0		Float	NO	
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	50e3	Hz	Float	NO	(0:FCenter*2)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth (required only when OrderType = Auto)	150e3	Hz	Float	NO	(PassBandwidth:FCenter*2)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	20		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]
Transform	S to Z domain transformation method: Bilinear, Impulse Invariance	Bilinear		Enumeration	NO	
UnderSampledModel	Default behavior when sampling rate is too small to represent the filter. This parameter is only used for simulation not for filter design tool: Model As Allpass, Error Out	Model As Allpass		Enumeration	NO	

Input Ports

Port	**Name**	**Signal Type**	**Optional**
1	input	envelope	NO

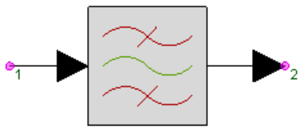
Output Ports

Port	**Name**	**Signal Type**	**Optional**
2	output	envelope	NO

Notes/Equations

1. BPF_SynchronouslyTuned implements a Bandpass Synchronously Tuned IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

BPF_Window (Bandpass Window Filter)



Description: Bandpass window-based linear phase FIR filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Window	Window function: Rectangular, Bartlett, Hann, Hamming, Blackman, Flat Top, Generalized Cosine, Ready, Kaiser, BlackmanHarris	Kaiser		Enumeration	NO	
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	50e3	Hz	Float	NO	(0:FCenter*2)
WindowParameter	Window parameter	1		Floating point array	NO	
Order	Filter order (number of taps - 1)	30		Integer	NO	[0:512]
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth (cutoff bandwidth = (PassBandwidth + StopBandwidth)/2)	100e3	Hz	Float	NO	(PassBandwidth:FCenter*2)
StopAtten	Stopband edge attenuation in dB	20		Float	NO	[3.0103:∞]
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

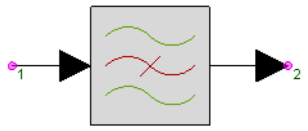
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BPF_Window implements a linear-phase Bandpass FIR filter using the Window design method.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *FIR Filter Design* (users) for the technical information about the design process.

BSF_Bessel (Bandstop Bessel Filter)



Description: Bandstop Bessel Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	200e3	Hz	Float	NO	(StopBandwidth:FCenter*2)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth (required only when OrderType = Auto)	50e3	Hz	Float	NO	(0:FCenter*2)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	20		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

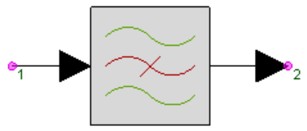
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BSF_Bessel implements a Bandstop Bessel IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

BSF_Butterworth (Bandstop Butterworth Filter)



Description: Bandstop Butterworth Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	100e3	Hz	Float	NO	(StopBandwidth:FCenter*2)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth (required only when OrderType = Auto)	50e3	Hz	Float	NO	(0:FCenter*2)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

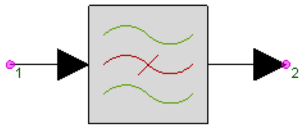
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BSF_Butterworth implements a Bandstop Butterworth IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

BSF_ChebyshevI (Bandstop ChebyshevI Filter)



Description: Bandstop Chebyshev I Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	100e3	Hz	Float	NO	(StopBandwidth:FCenter*2)
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth (required only when OrderType = Auto)	50e3	Hz	Float	NO	(0:FCenter*2)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

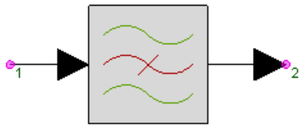
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BSF_ChebyshevI implements a Bandstop Chebyshev I IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

BSF_ChebyshevII (Bandstop ChebyshevII Filter)



Description: Bandstop Chebyshev II Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth (required only when OrderType = Auto)	100e3	Hz	Float	NO	(StopBandwidth:FCenter*2)
PassAtten	Passband edge attenuation in dB (required only when OrderType = Auto)	3		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth	50e3	Hz	Float	NO	(0:FCenter*2)
StopRipple	Stopband ripple in dB	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

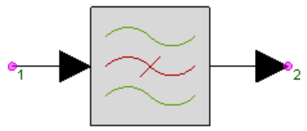
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BSF_ChebyshevII implements a Bandstop Chebyshev II IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

BSF_Elliptic (Bandstop Elliptic Filter)



Description: Bandstop Elliptic Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	100e3	Hz	Float	NO	(StopBandwidth:FCenter*2)
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth	50e3	Hz	Float	NO	(0:FCenter*2)
StopRipple	Stopband ripple in dB (required only when OrderType = Auto)	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

Output Ports

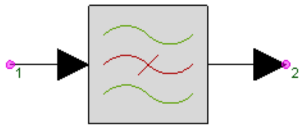
Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BSF_Elliptic implements a Bandstop Elliptic IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.

3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

BSF_ParksMcClellan (Bandstop ParksMcClellan Filter)



Description: Bandstop linear phase FIR filter based on Parks-McClellan design method
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	100e3	Hz	Float	NO	(StopBandwidth:FCenter*2)
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth	50e3	Hz	Float	NO	(0:FCenter*2)
StopRipple	Stopband ripple in dB	30		Float	NO	[3.0103:∞]
MaximumOrder	Maximum filter order for Parks-McClellan filter design	300		Integer	NO	
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

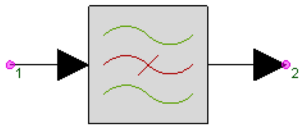
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BSF_ParksMcClellan implements a linear-phase Bandstop FIR filter using the Parks-McClellan design method.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *FIR Filter Design* (users) for the technical information about the design process.

BSF_SynchronouslyTuned (Bandstop SynchronouslyTuned Filter)



Description: Bandstop SynchronouslyTuned Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	200e3	Hz	Float	NO	(StopBandwidth:FCenter*2)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth (required only when OrderType = Auto)	50e3	Hz	Float	NO	(0:FCenter*2)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	14		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

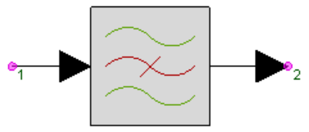
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BSF_SynchronouslyTuned implements a Bandstop Synchronously Tuned IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

BSF_Window (Bandstop Window Filter)



Description: Bandstop window-based linear phase FIR filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Window	Window function: Rectangular, Bartlett, Hann, Hamming, Blackman, Flat Top, Generalized Cosine, Ready, Kaiser, BlackmanHarris	Kaiser		Enumeration	NO	
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
FCenter	Center frequency	150e3	Hz	Float	NO	(0:∞)
PassBandwidth	Passband bandwidth	100e3	Hz	Float	NO	(0:FCenter*2)
WindowParameter	Window parameter	1		Floating point array	NO	
Order	Filter order (set to even number)	30		Integer	NO	[0:512]
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopBandwidth	Stopband bandwidth (cutoff bandwidth = (PassBandwidth + StopBandwidth)/2)	50e3	Hz	Float	NO	(0:PassBandwidth)
StopAtten	Stopband edge attenuation in dB	20		Float	NO	[3.0103:∞]
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

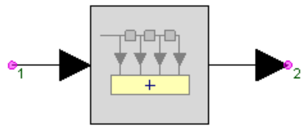
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. BSF_Window implements a linear-phase Bandstop FIR filter using the Window design method.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *FIR Filter Design* (users) for the technical information about the design process.

CustomFIR (Custom FIR Filter)



Description: Custom FIR filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Frequency	Frequency points in Hz	[0 20000 40000 60000 80000 100000 120000 140000 160000 180000 200000 220000 240000 260000 280000 300000 320000 340000 360000 380000 400000 420000 440000 460000 480000 500000]		Floating point array	NO
Magnitude	Magnitude response in dB at frequency points	[0 0 0 0 0 0 0 0 0 0 0 -5.333333333 -10.66666667 -16 -21.33333333 -26.66666667 -32 -37.33333333 -40 -40 -40 -40 -40 -40 -40 -40]		Floating point array	NO
Phase	Phase response in degrees at frequency points	[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]		Floating point array	NO
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO
MagTolerance	Magnitude error tolerance limit in dB	0.5		Float	NO
EnableFitFreqLimits	Enable limits for frequency fitting: NO, YES	NO		Enumeration	NO
LowerFitFreq	Lower fitting boundary within region [min(Frequency), max(Frequency)]	0	Hz	Float	NO
UpperFitFreq	Upper fitting boundary within region [min(Frequency), max(Frequency)]	100000	Hz	Float	NO
ForceLinearPhase	Force linear phase: NO, YES	NO		Enumeration	NO
ExtrapolationOption	Data extrapolation method outside data frequency range: Constant, versus freq	versus freq		Enumeration	NO
ExtrapolationRollOff	Additional rolloff (dB/octave) applied to data extrapolated outside data frequency range	0		Float	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

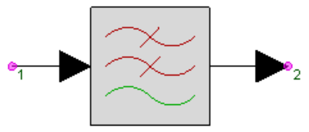
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. CustomFIR designs an FIR filter based on the frequency points and the associated magnitude and phase responses specified by the user in the *Frequency*, *Magnitude*, and *Phase* array parameters.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Custom FIR Design* (users) for details.

HBLPF_ParksMcClellan (Half-Band Lowpass ParksMcClellan Filter)



Description: Half-Band Lowpass linear phase FIR filter based on Parks-McClellan design method
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
PassFreq	Passband edge frequency	225e3	Hz	Float	NO	(0:∞)
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopRipple	Stopband ripple in dB	30		Float	NO	[3.0103:∞]
MaximumOrder	Maximum filter order for Parks-McClellan filter design	300		Integer	NO	
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

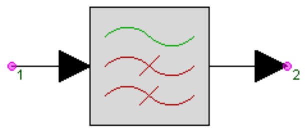
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. HBLPF_ParksMcClellan implements a linear-phase Half-Band Lowpass FIR filter using the Parks-McClellan design method.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *FIR Filter Design* (users) for the technical information about the design process.
5. HBLPF_ParksMcClellan is a half-band version of *LPF_ParksMcClellan* (algorithm) where (*PassFreq* + *StopFreq*)/2 = SampleRate/4 and 0 < *PassFreq* < SampleRate/4.

HPF Bessel (Highpass Bessel Filter)



Description: Highpass Bessel Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
PassFreq	Passband edge frequency	200e3	Hz	Float	NO	(StopFreq:∞)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency (required only when OrderType = Auto)	100e3	Hz	Float	NO	(0:∞)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	18		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

Output Ports

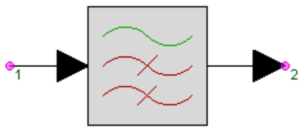
Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. HPF_Bessel implements a Highpass Bessel IIR filter.
2. This model consumes one sample from the input and produces one sample to the

output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

HPF_Butterworth (Highpass Butterworth Filter)



Description: Highpass Butterworth Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
PassFreq	Passband edge frequency	150e3	Hz	Float	NO	(StopFreq:∞)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency (required only when OrderType = Auto)	100e3	Hz	Float	NO	(0:∞)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

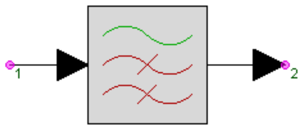
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. HPF_Butterworth implements a Highpass Butterworth IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

HPF_ChebyshevI (Highpass ChebyshevI Filter)



Description: Highpass Chebyshev I Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
PassFreq	Passband edge frequency	150e3	Hz	Float	NO	(StopFreq:∞)
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency (required only when OrderType = Auto)	100e3	Hz	Float	NO	(0:∞)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

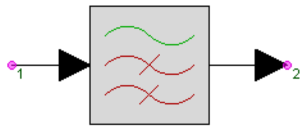
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. HPF_ChebyshevI implements a Highpass Chebyshev I IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

HPF_ChebyshevII (Highpass ChebyshevII Filter)



Description: Highpass Chebyshev II Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
PassFreq	Passband edge frequency (required only when OrderType = Auto)	150e3	Hz	Float	NO	(StopFreq:∞)
PassAtten	Passband edge attenuation in dB (required only when OrderType = Auto)	3		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency	100e3	Hz	Float	NO	(0:∞)
StopRipple	Stopband ripple in dB	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

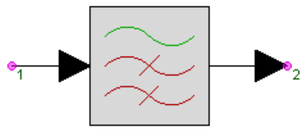
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. HPF_ChebyshevII implements a Highpass Chebyshev II IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

HPF_Elliptic (Highpass Elliptic Filter)



Description: Highpass Elliptic Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
PassFreq	Passband edge frequency	150e3	Hz	Float	NO	(StopFreq:∞)
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency	100e3	Hz	Float	NO	(0:∞)
StopRipple	Stopband ripple in dB (required only when OrderType = Auto)	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

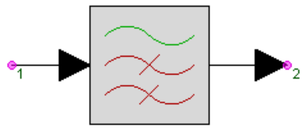
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. HPF_Elliptic implements a Highpass Elliptic IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

HPF_ParksMcClellan (Highpass ParksMcClellan Filter)



Description: Highpass linear phase FIR filter based on Parks-McClellan design method
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
PassFreq	Passband edge frequency	150e3	Hz	Float	NO	(0:∞)
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency	100e3	Hz	Float	NO	(PassFreq:∞)
StopRipple	Stopband ripple in dB	30		Float	NO	[3.0103:∞]
MaximumOrder	Maximum filter order for Parks-McClellan filter design	300		Integer	NO	
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

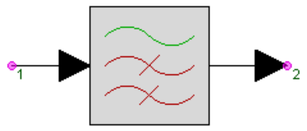
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. HPF_ParksMcClellan implements a linear-phase Highpass FIR filter using the Parks-McClellan design method.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *FIR Filter Design* (users) for the technical information about the design process.

HPF_SynchronouslyTuned (Highpass SynchronouslyTuned Filter)



Description: Highpass Synchronously Tuned Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
PassFreq	Passband edge frequency	250e3	Hz	Float	NO	(StopFreq:∞)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency (required only when OrderType = Auto)	100e3	Hz	Float	NO	(0:∞)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	20		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

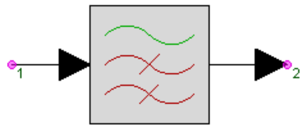
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. HPF_SynchronouslyTuned implements a Highpass Synchronously Tuned IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

HPF_Window (Highpass Window Filter)



Description: Highpass window-based linear phase FIR filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Window	Window function: Rectangular, Bartlett, Hann, Hamming, Blackman, Flat Top, Generalized Cosine, Ready, Kaiser, BlackmanHarris	Kaiser		Enumeration	NO	
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
PassFreq	Passband edge frequency	150e3	Hz	Float	NO	(0:∞)
WindowParameter	Window parameter	1		Floating point array	NO	
Order	Filter order (number of taps - 1)	30		Integer	NO	[0:512]
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency (cutoff frequency = (PassFreq + StopFreq)/2)	100e3	Hz	Float	NO	(0:PassFreq)
StopAtten	Stopband edge attenuation in dB	20		Float	NO	[3.0103:∞]
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

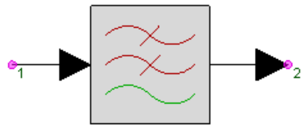
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. HPF_Window implements a linear-phase Highpass FIR filter using the Window design method.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *FIR Filter Design* (users) for the technical information about the design process.

LPF_Bessel (Lowpass Bessel Filter)



Description: Lowpass Bessel Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
PassFreq	Passband edge frequency	100e3	Hz	Float	NO	(0:∞)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency (required only when OrderType = Auto)	200e3	Hz	Float	NO	(PassFreq:∞)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	18		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]
Transform	S to Z domain transformation method: Bilinear, Impulse Invariance	Bilinear		Enumeration	NO	
UnderSampledModel	Default behavior when sampling rate is too small to represent the filter. This parameter is only used for simulation not for filter design tool: Model As Allpass, Error Out	Model As Allpass		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

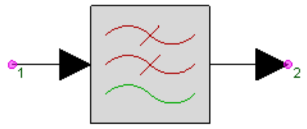
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. LPF_Bessel implements a Lowpass Bessel IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

LPF_Butterworth (Lowpass Butterworth Filter)



Description: Lowpass Butterworth Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
PassFreq	Passband edge frequency	100e3	Hz	Float	NO	(0:∞)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency (required only when OrderType = Auto)	150e3	Hz	Float	NO	(PassFreq:∞)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]
Transform	S to Z domain transformation method: Bilinear, Impulse Invariance	Bilinear		Enumeration	NO	
UnderSampledModel	Default behavior when sampling rate is too small to represent the filter. This parameter is only used for simulation not for filter design tool: Model As Allpass, Error Out	Model As Allpass		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

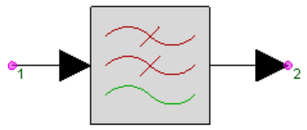
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. LPF_Butterworth implements a Lowpass Butterworth IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

LPF_ChebyshevI (Lowpass ChebyshevI Filter)



Description: Lowpass Chebyshev I Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
PassFreq	Passband edge frequency	100e3	Hz	Float	NO	(0:∞)
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency (required only when OrderType = Auto)	150e3	Hz	Float	NO	(PassFreq:∞)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]
Transform	S to Z domain transformation method: Bilinear, Impulse Invariance	Bilinear		Enumeration	NO	
UnderSampledModel	Default behavior when sampling rate is too small to represent the filter. This parameter is only used for simulation not for filter design tool: Model As Allpass, Error Out	Model As Allpass		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

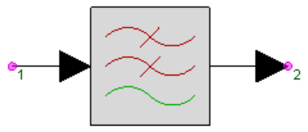
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. LPF_ChebyshevI implements a Lowpass Chebyshev I IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

LPF_ChebyshevII (Lowpass ChebyshevII Filter)



Description: Lowpass Chebyshev II Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
PassFreq	Passband edge frequency (required only when OrderType = Auto)	100e3	Hz	Float	NO	(0:∞)
PassAtten	Passband edge attenuation in dB (required only when OrderType = Auto)	3		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency	150e3	Hz	Float	NO	(PassFreq:∞)
StopRipple	Stopband ripple in dB	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]
UnderSampledModel	Default behavior when sampling rate is too small to represent the filter. This parameter is only used for simulation not for filter design tool: Model As Allpass, Error Out	Model As Allpass		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

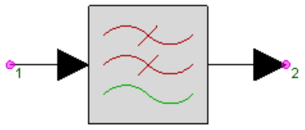
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. LPF_ChebyshevII implements a Lowpass Chebyshev II IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

LPF_Edge (Lowpass Edge Filter)



Description: EDGE pulse shaping lowpass filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
SymbolRate	Symbol rate (impulse time = 5 * symbol period)	1625000/6	Hz	Float	NO	(0:∞)
Window	Window function: Rectangular, Bartlett, Hann, Hamming, Blackman, Flat Top	Rectangular		Enumeration	NO	
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

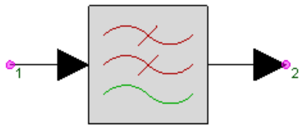
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. LPF_Edge implements a lowpass EDGE pulse shaping filter.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *FIR Filter Design* (users) for the technical information about the design process.

LPF_Elliptic (Lowpass Elliptic Filter)



Description: Lowpass Elliptic Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
PassFreq	Passband edge frequency	100e3	Hz	Float	NO	(0:∞)
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency	150e3	Hz	Float	NO	(PassFreq:∞)
StopRipple	Stopband ripple in dB (required only when OrderType = Auto)	50		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]
UnderSampledModel	Default behavior when sampling rate is too small to represent the filter. This parameter is only used for simulation not for filter design tool: Model As Allpass, Error Out	Model As Allpass		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

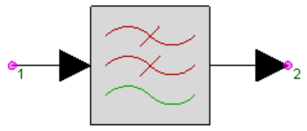
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. LPF_Elliptic implements a Lowpass Elliptic IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

LPF_Gaussian (Lowpass Gaussian Filter)



Description: Gaussian lowpass filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
PassFreq	Passband edge frequency	100e3	Hz	Float	NO	(0:∞)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
LengthOption	Filter length option (Auto: 1.2 / 3db frequency): Auto, Number of Taps, Impulse Time Duration	Auto		Enumeration	NO	
Length	Filter length (based on LengthOption, specify number of taps, or impulse time duration; delay = length/2)	0		Float	NO	[0:∞)
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

Output Ports

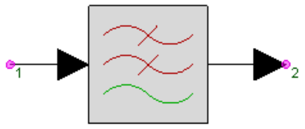
Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. The delay introduced by the filter is approximately equal to:

$$Delay = \frac{0.6}{\sqrt{3.0/(PassAtten)} \times PassFreq}$$

LPF_ParksMcClellan (Lowpass ParksMcClellan Filter)



Description: Lowpass linear phase FIR filter based on Parks-McClellan design method
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
PassFreq	Passband edge frequency	100e3	Hz	Float	NO	(0:∞)
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency	150e3	Hz	Float	NO	(PassFreq:∞)
StopRipple	Stopband ripple in dB	30		Float	NO	[3.0103:∞]
MaximumOrder	Maximum filter order for Parks-McClellan filter design	300		Integer	NO	
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

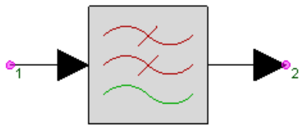
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. LPF_ParksMcClellan implements a linear-phase Lowpass FIR filter using the Parks-McClellan design method.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *FIR Filter Design* (users) for the technical information about the design process.

LPF_RaisedCosine (Lowpass Raised Cosine Filter)



Description: Raised cosine lowpass filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
SymbolRate	Symbol rate (passband edge frequency = SymbolRate/2)	100e3	Hz	Float	NO	(0:∞)
RollOff	Roll-off factor (between 0 and 1)	0.5		Float	NO	[0:1]
SquareRoot	Square root option: NO, YES	NO		Enumeration	NO	
PulseEqualization	Pulse equalization option: NO, YES	NO		Enumeration	NO	
LengthOption	Filter length option (Auto: 8 * symbol period): Auto, Number of Taps, Number of Symbols, Impulse Time Duration	Auto		Enumeration	NO	
Length	Filter length (length = 2*delay) (based on LengthOption, specify number of taps, number of symbols, or impulse time duration)	0		Float	NO	[0:∞)
Window	Window function: Rectangular, Bartlett, Hann, Hamming, Blackman, Flat Top	Rectangular		Enumeration	NO	
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

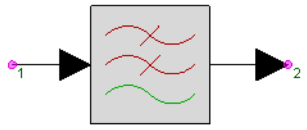
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. LPF_RaisedCosine implements a Lowpass Raised Cosine FIR filter.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *FIR Filter Design* (users) for the technical information about the design process.

LPF_SynchronouslyTuned (Lowpass SynchronouslyTuned Filter)



Description: Lowpass Synchronously Tuned Filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Loss	Magnitude loss in dB	0		Float	NO	
PassFreq	Passband edge frequency	100e3	Hz	Float	NO	(0:∞)
PassAtten	Passband edge attenuation in dB	3		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency (required only when OrderType = Auto)	250e3	Hz	Float	NO	(PassFreq:∞)
StopAtten	Stopband edge attenuation in dB (required only when OrderType = Auto)	20		Float	NO	(3.0103:∞)
OrderType	Order specification: Auto, User Defined	User Defined		Enumeration	NO	
Order	User defined order for the lowpass prototype analog filter	5		Integer	NO	[0:30]
Transform	S to Z domain transformation method: Bilinear, Impulse Invariance	Bilinear		Enumeration	NO	
UnderSampledModel	Default behavior when sampling rate is too small to represent the filter. This parameter is only used for simulation not for filter design tool: Model As Allpass, Error Out	Model As Allpass		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

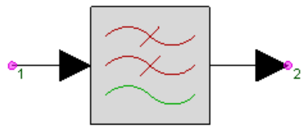
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. LPF_SynchronouslyTuned implements a Lowpass Synchronously Tuned IIR filter.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *IIR Filter Design* (users) for the technical information about the design process.

LPF_Window (Lowpass Window Filter)



Description: Lowpass window-based linear phase FIR filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Window	Window function: Rectangular, Bartlett, Hann, Hamming, Blackman, Flat Top, Generalized Cosine, Ready, Kaiser, BlackmanHarris	Kaiser		Enumeration	NO	
Loss	Magnitude loss in dB	0		Float	NO	[0:∞)
PassFreq	Passband edge frequency	100e3	Hz	Float	NO	(0:∞)
WindowParameter	Window parameter	1		Floating point array	NO	
Order	Filter order (number of taps - 1)	30		Integer	NO	[0:512]
PassRipple	Passband ripple in dB	1		Float	NO	[0.01:3.0103]
StopFreq	Stopband edge frequency (cutoff frequency = (PassFreq + StopFreq)/2)	150e3	Hz	Float	NO	(PassFreq:∞)
StopAtten	Stopband edge attenuation in dB	20		Float	NO	[3.0103:∞]
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

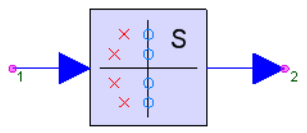
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. LPF_Window implements a linear-phase Lowpass FIR filter using the Window design method.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Please refer to *FIR Filter Design* (users) for the technical information about the design process.

SDomainSystem (S Domain System Filter)



Description: S domain system
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *SDomainIIR Part* (algorithm), *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Factor	H(s) = Factor * ((s - z1) * (s - z2) * ...) / ((s - p1) * (s - p2) * ...)	1.1616128054708951e+029		Float	NO
RealPoles	Real poles	[-650148.07080641726]		Floating point array	NO
ComplexConjugatePoles	Complex conjugate poles (specify only one for a complex conjugate pair)	[-200906.80273926951 + 618327.55929716607j, -525980.83814247814 + 382147.44782641466j]		Complex array	NO
RealZeros	Real zeros	[]		Floating point array	NO
ComplexConjugateZeros	Complex conjugate zeros (specify only one for a complex conjugate pair)	[]		Complex array	NO
FreqUnit	Frequency unit for S domain poles and zeros: Radians Per Second, Hz	Radians Per Second		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

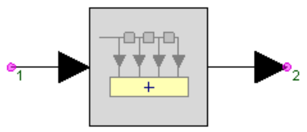
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. SDomainSystem designs an IIR filter based on the S domain poles and zeros specified by the users.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *S Domain Design* (users) for the technical information about the design process.
4. Suppose *Factor* = C, *RealPoles* = [p_r], *ComplexConjugatePoles* = [p_{c1} p_{c2}], *RealZeros* = [z_r], *ComplexConjugateZeros* = [z_{c1} z_{c2}], then the S domain transfer function is
$$H(s) = C\frac{(s-z_r)(s-z_{c1})(s-z_{c1}^*)(s-z_{c2})(s-z_{c2}^*)}{(s-p_r)(s-p_{c1})(s-p_{c1}^*)(s-p_{c2})(s-p_{c2}^*)}$$
5. For *ComplexConjugatePoles* and *ComplexConjugateZeros*, only specify one for each complex conjugate pair.
6. SDomainSystem applies Bilinear Transform to convert S-domain transfer function H(s) to Z-domain IIR transfer function H(z).

ZDomainFIR (Z Domain FIR filter)



Description: Z domain FIR filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Coefficients	Filter tap values	[-0.040609, -0.001628, 0.17853, 0.37665, 0.37665, 0.17853, -0.001628, -0.040609]		Floating point array	NO	
Interpolation	Interpolation (up sampling) factor	1		Integer	NO	[1:∞)
Decimation	Decimation (down sampling) factor	1		Integer	NO	[1:∞)
DecimationPhase	Decimation (down sampling) phase	0		Integer	NO	[0:_D_ecimation-1]
InterpolationScaling	Gain adjustment for interpolation: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

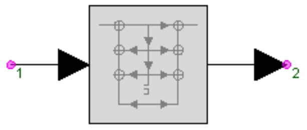
Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. ZDomainFIR is an FIR filter with the coefficients specified by the users.
2. This model consumes *Decimation* number of samples from the input and produces *Interpolation* number of samples to the output in every execution.
3. Please refer to *Common Filter Parameters* (algorithm) for further parameter details.
4. Suppose *Coefficients* = [b_0 b_1 … b_N], then $H(z) = b_0 + b_1 z^{-1} + \ldots + b_N z^{-N}$.

ZDomainIIR (Z Domain IIR filter)



Description: Z domain IIR filter
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Numerator	Numerator coefficients. Single section: specify a real array of numerator coefficients. Multiple sections: specify a cell array of real arrays (each real array contains numerator coefficients of a section).	[0.5, 0.25, 0.1]		None	NO
Denominator	Denominator coefficients. Single section: specify a real array of denominator coefficients. Multiple sections: specify a cell array of real arrays (each real array contains denominator coefficients of a section).	[1, 0.5, 0.3]		None	NO
Structure	Transfer function structure for multiple sections.: Cascade Form, Parallel Form	Cascade Form		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. ZDomainIIR is an IIR filter with coefficients specified by the users.
2. This model consumes one sample from the input and produces one of sample to the output in every execution.
3. Suppose *Numerator* = [b_{00} b_{01} b_{02}] and *Denominator* = [a_{00} a_{01} a_{02}], then
$$H(z) = \frac{b_{00} + b_{01}z^{-1} + b_{02}z^{-2}}{a_{00} + a_{01}z^{-1} + a_{02}z^{-2}}$$
4. Suppose *Numerator* = { [b_{00} b_{01} b_{02}], [b_{10} b_{11} b_{12}] } and *Denominator* = { [a_{00}

a_{01} a_{02}], [a_{10} a_{11} a_{12}] }.

When *Structure* = Cascade Form,

$$H(z) = \frac{b_{00} + b_{01}z^{-1} + b_{02}z^{-2}}{a_{00} + a_{01}z^{-1} + a_{02}z^{-2}} \times \frac{b_{10} + b_{11}z^{-1} + b_{12}z^{-2}}{a_{10} + a_{11}z^{-1} + a_{12}z^{-2}}$$

.

When *Structure* = Parallel Form,

$$H(z) = \frac{b_{00} + b_{01}z^{-1} + b_{02}z^{-2}}{a_{00} + a_{01}z^{-1} + a_{02}z^{-2}} + \frac{b_{10} + b_{11}z^{-1} + b_{12}z^{-2}}{a_{10} + a_{11}z^{-1} + a_{12}z^{-2}}$$

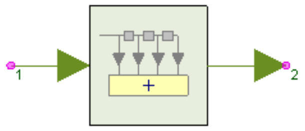
FIR_Cx Part

Categories: C++ Code Generation , *Filters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
FIR_Cx (algorithm)	std::complex<double> FIR

FIR_Cx (Complex FIR Filter)



Description: Complex FIR Filter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *FIR Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Taps	Filter tap values	[-0.040609, -0.001628, 0.17853, 0.37665, 0.37665, 0.17853, -0.001628, -0.040609]		Complex array	NO
Decimation	Decimation ratio	1		Integer	NO
DecimationPhase	Decimation phase	0		Integer	NO
Interpolation	Interpolation ratio	1		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	complex	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	complex	NO

Notes/Equations

1. The FIR_Cx component implements a complex-valued finite-impulse response filter with multi-rate capability.
2. For every Decimation number of input samples, Interpolation number of filtered values are output.
3. The filter coefficients are specified by the Taps parameter.
4. FIR efficiently implements sample rate changes. When the Decimation ratio is ≥1, the filter behaves as if it were followed by a *DownSample* (algorithm) part. When the Interpolation ratio is set, the filter behaves as if it were preceded by an *UpSample* (algorithm) part. The implementation is much more efficient, because a polyphase structure is used internally, thereby avoiding unnecessary use of memory and multiplications by 0. Arbitrary sample rate conversions by rational factors are accomplished this way.
5. The DecimationPhase parameter is equivalent to the Phase parameter of the *DownSample* (algorithm) part. When decimating, output samples are conceptually discarded. The polyphase structure does not calculate the discarded samples. To decimate by three, only one of every three outputs is selected. The DecimationPhase parameter determines which of these is selected. If DecimationPhase is 0 (default), the earliest outputs of the decimation block are decimated.
6. Avoid aliasing when designing a multi-rate filter. The filter sample rate is the product of the Interpolation parameter and the input sample rate or equivalently the product of the Decimation parameter and the output sample rate.
7. The default tap coefficients correspond to an eighth-order, equiripple, linear-phase, lowpass filter. The cutoff frequency is approximately one-third of the Nyquist frequency.

See:
FIR (algorithm)
FIR_Fxp (hardware)

References

1. F. J. Harris, "Multirate FIR Filters for Interpolating and Desampling," *Handbook of*

Digital Signal Processing, Academic Press, 1987.

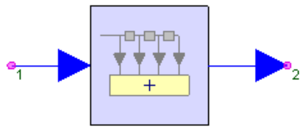
FIR Part

Categories: *C++ Code Generation* (algorithm), *Filters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
FIR (algorithm)	FIR Filter
FIR_Cx (algorithm)	Complex FIR Filter
FIR_Fxp (hardware)	Fixed Point Finite Impulse Response Filter

FIR (FIR Filter)



Description: FIR Filter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *FIR Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Taps	Filter tap values	[-0.040609, -0.001628, 0.17853, 0.37665, 0.37665, 0.17853, -0.001628, -0.040609]		Floating point array	NO
Decimation	Decimation ratio	1		Integer	NO
DecimationPhase	Decimation phase	0		Integer	NO
Interpolation	Interpolation ratio	1		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. FIR implements a finite-impulse response filter with multi-rate capability.
2. For every Decimation number of input samples, Interpolation number of filtered values are output.
3. The Taps parameter specifies the filter coefficients.
4. FIR efficiently implements sample rate changes. When the Decimation ratio is ≥1, the filter behaves as if it were followed by a *DownSample* (algorithm) part. When the Interpolation ratio is set, the filter behaves as if it were preceded by an *UpSample* (algorithm) part. The implementation is much more efficient, because a polyphase structure is used internally, thereby avoiding unnecessary use of memory and multiplications by 0. Arbitrary sample rate conversions by rational factors are accomplished this way.
5. The DecimationPhase parameter is equivalent to the Phase parameter of the *DownSample* (algorithm) part. When decimating, output samples are conceptually discarded. The polyphase structure does not calculate the discarded samples. To decimate by three, only one of every three outputs is selected. The DecimationPhase parameter determines which of these is selected. If DecimationPhase is 0 (default), the earliest outputs of the decimation block are decimated.
6. Avoid aliasing when designing a multi-rate filter. The filter sample rate is the product of the Interpolation parameter and the input sample rate or equivalently the product of the Decimation parameter and the output sample rate.
7. The default tap coefficients correspond to an eighth-order, equiripple, linear-phase, lowpass filter. The cutoff frequency is approximately one-third of the Nyquist frequency.

See:
FIR_Cx (algorithm)
FIR_Fxp (hardware)
ResamplerRC (algorithm)

References

1. F. J. Harris, "Multirate FIR Filters for Interpolating and Desampling," *Handbook of Digital Signal Processing*, Academic Press, 1987.
2. A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall: Englewood Cliffs, NJ, 1989.
3. P. P. Vaidyanathan, "Multirate Digital Filters, Filter Banks, Polyphase Networks, and Applications: A Tutorial," *Proc. of the IEEE*, vol. 78, no. 1, pp. 56-93, Jan. 1990.

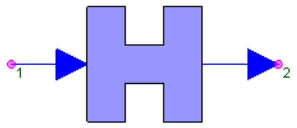
Hilbert Part

Categories: *C++ Code Generation* (algorithm), *Filters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Hilbert (algorithm)	Hilbert Transformer

Hilbert (Hilbert Transformer)



Description: Hilbert Transformer
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Hilbert Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
N	Number of taps in the Hilbert filter	64		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. This component approximates the Hilbert transform of the input signal by using an FIR filter.
2. For every input, there is a transformed output.
3. The response is truncated symmetrically at $-N/2$ and $N/2$ samples [1]. For higher accuracy and smaller delay, it may be necessary to use the Parks-McClellan algorithm [2] to design a custom Hilbert transform filter [1,3].
4. The Hilbert transform requires an infinite length set of FIR tap coefficients for accurate representation. This model approximates the Hilbert transform with a finite list of FIR taps. For practical accuracy, it is recommended $N \geq 64$.

See:
FIR (algorithm)

References

1. A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall: Englewood Cliffs, NJ, 1989.
2. T. W. Parks and J. H. McClellan, "Chebyshev Approximation for Nonrecursive Digital Filters With Linear Phase," *IEEE Trans. on Circuit Theory*, vol. 19, no. 2, pp. 189-194, March 1972.
3. L. R. Rabiner, J. H. McClellan, and T. W. Parks, "FIR Digital Filter Design Techniques Using Weighted Chebyshev Approximation," *Proc. of the IEEE*, vol. 63, no. 4, pp. 595-610, April 1975.

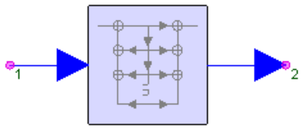
IIR Part

Categories: *C++ Code Generation* (algorithm), *Filters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
IIR (algorithm)	IIR Filter
IIR_Cx (algorithm)	Complex IIR Filter

IIR (IIR Filter)



Description: IIR Filter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *IIR Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Gain	Gain Value	1		Float	YES
Numerator	Numerator coefficients	[0.5, 0.25, 0.1]		Floating point array	NO
Denominator	Denominator coefficients	[1, 0.5, 0.3]		Floating point array	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

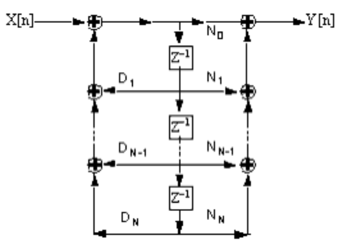
Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. IIR implements an infinite impulse response filter of arbitrary order in a direct form II as shown in IIR Filter Structure.
2. For every input, one output is generated.
3. The parameters specify *H(z)*, the Z-transform of an impulse response *h(n)*. The output of IIR is the convolution of the input with *h(n)*.
   The transfer function is of the form
   $$H(z) = G\frac{N(z^{-1})}{D(z^{-1})}$$
   where:
   Gain specifies *G*.
   Numerator and Denominator specify $N(z^{-1})$ and $D(z^{-1})$, respectively. Both arrays start with the constant term of the polynomial and decrease in powers of *z* or increase in powers of 1/*z*. The constant term of *D* is not omitted.

   IIR Filter Structure

4. Numerator and Denominator are array values.
5. The numerical noise originating from finite precision increases with the filter order. To minimize this distortion, expand the filter into a parallel or cascade form.

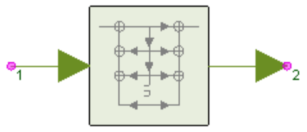
See:
Biquad (algorithm)
BiquadCascade (algorithm)

IIR_Cx (algorithm)

References

1. A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall: Englewood Cliffs, NJ, 1989.

IIR_Cx (Complex IIR Filter)



Description: Complex IIR Filter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *IIR Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Gain	Gain Value	1		Complex number	YES
Numerator	Numerator coefficients	[0.5, 0.25, 0.1]		Complex array	NO
Denominator	Denominator coefficients	[1, 0.5, 0.3]		Complex array	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	complex	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	complex	NO

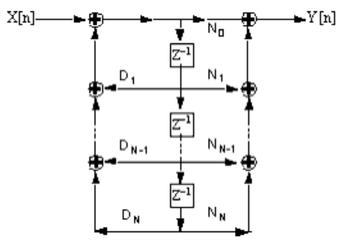
Notes/Equations

1. IIR_Cx implements a complex infinite impulse response (IIR) filter of arbitrary order in a direct form II realization.
2. For every input, one output is generated.
3. The parameters specify *H(z)*, the Z-transform of an impulse response *h(n)*. The output is the convolution of the input with *h(n)*. The transfer function is of the form

$$H(z) = G\frac{N(z^{-1})}{D(z^{-1})}$$

   where:
   Gain specifies *G*.
   Numerator and Denominator specify $N(z^{-1})$ and $D(z^{-1})$, respectively. Both arrays start with the constant term of the polynomial and decrease in powers of *z* or increase in powers of 1/*z*. The constant term of *D* is not omitted.

   IIR Filter Structure



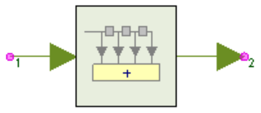
4. Numerator and Denominator are array values.
5. The numerical noise originating from finite precision increases with the filter order. To minimize this distortion, expand the filter into a parallel or cascade form.

See:
IIR (algorithm)

References

1. A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall: Englewood Cliffs, NJ, 1989.

FIR_CX



Description: Complex FIR Filter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *FIR Part* (algorithm)

Model Parameters

Name	**Description**	**Default**	**Units**	**Type**	**Runtime Tunable**
Taps	Filter tap values	[-0.040609, -0.001628, 0.17853, 0.37665, 0.37665, 0.17853, -0.001628, -0.040609]		Complex array	NO
Decimation	Decimation ratio	1		Integer	NO
DecimationPhase	Decimation phase	0		Integer	NO
Interpolation	Interpolation ratio	1		Integer	NO

Input Ports

Port	**Name**	**Signal Type**	**Optional**
1	input	complex	NO

Output Ports

Port	**Name**	**Signal Type**	**Optional**
2	output	complex	NO

Notes/Equations

References

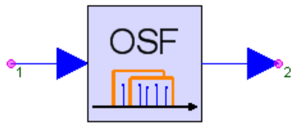
OSF Part

Categories: *Filters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
OSF (algorithm)	Order Statistic Filter

OSF (Order Statistic Filter)



Description: Order Statistic Filter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *OSF Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
N	Size of sliding window	3		Integer	NO	[1:∞)
Percentile	Ranking percentile (0 percent is the minimum)	50		Float	YES	[0:100]

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. OSF implements a order statistic filter.
2. For every input, one output is generated.
3. From a window of the most recent N input values, a value is chosen for output. If each of N inputs are ranked from the lowest (0 percentile) to the highest value (100 percentile), the value ranked closest to the Percentile parameter is output.
4. The default part outputs the median value of the most recent 3 inputs.

See:
SlidWinAvg (algorithm)

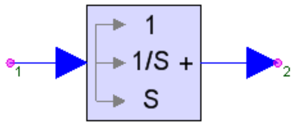
PID Part

Categories: *Filters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
PID (algorithm)	Proportional-Integral-Derivative Controller

PID (Proportional-Integral-Derivative Controller)



Description: Proportional-Integral-Derivative Controller
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *PID Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
ProportionalGain	Proportional gain	1		Float	YES	[(-∞:∞)
IntegratorGain	Integral gain	1		Float	YES	(-∞:∞)
DerivativeGain	Derivative gain	1		Float	YES	(-∞:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. PID outputs the weighted sum of the input, the integral of the input, and the derivative of the input.
2. For every input, one output is generated.
3. Both integral and derivative are zero for the first output. The derivative and the increment for the integral require only the most recent input samples.

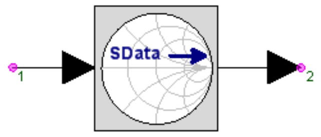
SData Part

Categories: *Analog/RF* (algorithm), *Filters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SData (algorithm)	Reading and Simulation of S-Parameter Data

SData



Description: Reading and Simulation of S-Parameter Data
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *SData Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
OutputPort	Reflected wave port (row index for the desired S-Parameter in the scattering matrix)	2		Integer	NO	[1:∞)
InputPort	Incident wave port (column index for the desired S-Parameter in the scattering matrix)	1		Integer	NO	[1:∞)
DataSource	S-Parameter data location: TouchstoneFile, Dataset	TouchstoneFile		Enumeration	NO	
DataFileName	Touchstone data file name (e.g. TwoPort.s2p)			Filename	NO	
DatasetName	Dataset name			Text	NO	
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO	
MagTolerance	Magnitude error tolerance limit in dB	0.5		Float	NO	
EnableFitFreqLimits	Enable limits for frequency fitting: NO, YES	NO		Enumeration	NO	
LowerFitFreq	Lower fitting boundary within region [min(Frequency), max(Frequency)]	0	Hz	Float	NO	
UpperFitFreq	Upper fitting boundary within region [min(Frequency), max(Frequency)]	100000	Hz	Float	NO	
ForceLinearPhase	Force linear phase: NO, YES	NO		Enumeration	NO	
ExtrapolationOption	Data extrapolation method outside data frequency range: Constant, versus freq	versus freq		Enumeration	NO	
ExtrapolationRollOff	Additional rolloff (dB/octave) applied to data extrapolated outside data frequency range	0		Float	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	envelope	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	envelope	NO

Notes/Equations

1. The SData model can be used to simulate S-Parameter data in the time domain Data Flow simulation engine.
2. Only the data for a single pair of ports can be simulated. The *OutputPort* and *InputPort* parameters specify the pair of ports whose S-Parameter data will be used.
3. The *DataSource* parameter selects whether the S-Parameter data resides in a *TouchstoneFile* file or in a *Dataset* in the workspace.
    - When *Touchstone* is selected, the name of the file is specified in the *DataFileName* parameter.

- When *Dataset* is selected, the name of the dataset is specified in the *DatasetName* parameter (to import S-Parameter data from an S-Data file or a Touchstone file to a dataset go to the File menu and select Import -> S-Data File ...)

4. The S-Parameter data is converted to an equivalent causal time domain impulse response of finite length (FIR filter) that is used by the Data Flow simulator.
5. For details on the parameters shown under *ShowAdvancedParams* refer to *Custom FIR Design* (users).

References

1. The ***SData Demo*** **(examples)** example demonstrates how to use SData part.

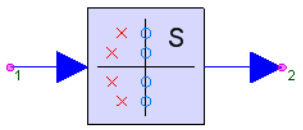
SDomainIIR Part

Categories: *C++ Code Generation* (algorithm), *Filters* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SDomainIIR (algorithm)	S-domain IIR filter using bilinear transform and cascade biquad structure
SDomainSystem (algorithm)	S domain system

SDomainIIR (S DomainIIR)



Description: S-domain IIR filter using bilinear transform and cascade biquad structure
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *SDomainIIR Part* (algorithm), *Filter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
SampleRate	Sampling rate	1e6		Float	NO
Factor	H(s) = Factor * ((s - z1) * (s - z2) * ...) / ((s - p1) * (s - p2) * ...)	1.1616128054708951e+029		Float	NO
RealPoles	Real Poles	[-650148.07080641726]		None	NO
ComplexConjugatePoles	Complex conjugate poles (specify only one for a complex conjugate pair)	[-200906.80273926951 + 618327.55929716607j, -525980.83814247814 + 382147.44782641466j]		None	NO
RealZeros	Real zeros	[]		None	NO
ComplexConjugateZeros	Complex conjugate zeros (specify only one for a complex conjugate pair)	[]		None	NO
FreqUnit	Frequency unit: Radians per second, Hz	Radians per second		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. SDomainIIR designs an IIR biquad-cascade filter based on the S domain poles and zeros specified by the users.
2. This model consumes one sample from the input and produces one sample to the output in every execution.
3. Please refer to *S Domain Design* (users) for the technical information about the design process.
4. Suppose *Factor* = C, *RealPoles* = [p_r], *ComplexConjugatePoles* = [p_{c1} p_{c2}], *RealZeros* = [z_r], *ComplexConjugateZeros* = [z_{c1} z_{c2}], then the S domain transfer function is
$$H(s) = C\frac{(s-z_r)(s-z_{c1})(s-z_{c1}^*)(s-z_{c2})(s-z_{c2}^*)}{(s-p_r)(s-p_{c1})(s-p_{c1}^*)(s-p_{c2})(s-p_{c2}^*)}$$
5. For *ComplexConjugatePoles* and *ComplexConjugateZeros*, only specify one for each complex conjugate pair.

6. SDomainIIR applies Bilinear Transform to convert S-domain transfer function H(s) to Z-domain IIR transfer function H(z).
7. SDomainIIR is an untimed model. The SampleRate parameter specifies the sampling rate to characterize the S-domain transfer function and to transform it into Z-domain transfer function. The difference between SDomainIIR and *SDomainSystem* (algorithm) is that SDomainSystem uses simulation sampling rate to characterize the S-doamin transfer function, but SDomainIIR uses SampleRate parameter instead.
8. SDomainIIR supports *C++ code generation* (algorithm). In the generated C++ code, the SDomainIIR is re-characterized based on the SampleRate parameter.

IBIS-AMI_Transceivers

- *BlindDFE Part* (algorithm)
- *BlindFFE Part* (algorithm)
- *CDR Part* (algorithm)
- *ClockTimes Part* (algorithm)
- *Coder64b66b Part* (algorithm)
- *Coder8b10b Part* (algorithm)
- *Decoder64b66b Part* (algorithm)
- *Decoder8b10b Part* (algorithm)
- *DFE Part* (algorithm)
- *FFE Part* (algorithm)
- *PhaseDetector Part* (algorithm)
- *PulseShaping Part* (algorithm)
- *SDomainIIR Part* (algorithm)
- *TimeResponseFIR Part* (algorithm)
- *VCO Part* (algorithm)

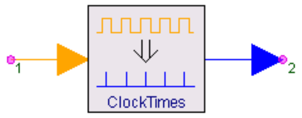
ClockTimes Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
ClockTimes (algorithm)	Clock signal to clock times converter

ClockTimes



Description: Clock signal to clock times converter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *ClockTimes Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
SampleInterval	Sample interval (time step between adjacent clock samples)	1e-10	s	Float	NO
ClockEdge	Clock edge option for clock times: Negative clock edge, Positive clock edge	Negative clock edge		Enumeration	NO
Offset	Offset for clock times	0	s	Float	NO

Input Ports

Port	Name	Signal Type	Optional
1	clock	int	NO

Output Ports

Port	Name	Signal Type	Optional
2	time	real	NO

Notes and Equations

1. The ClockTimes model converts clock signal to clock times.
2. According to IBIS Version 5.0 specification, **AMI_GetWave** function can return clock times that represent the times at which clock signal at the output of the clock recovery loop crosses the logic threshold. The input samples are assumed to be sampled at exactly one half clock period after a clock time.
3. This model consumes one sample from the **clock** port and produces one sample to the **time** port in every execution.
4. Parameter **SampleInterval** specifies the sample interval (1 / sample rate) of the input clock signal.
5. Parameter **ClockEdge** specifies which clock edge to be timed.
6. Parameter **Offset** specifies the offset to be used for clock times.
7. Let *c[n]* represent the input clock samples. A positive clock edge occurs at sample instance *n* if *c[n] > 0* and *c[n-1] <= 0*. A negative clock edge occurs at sample instance *n* if *c[n] <= 0* and *c[n-1] > 0*.
8. For samples that do not belong valid clock time instances (positive or negative clock edges), the value of the output sample is -1. AMI_GetWave generated by SystemVue will remove negative-valued clock times. For more information, please refer to *Understanding AMI Model Generation* (users).

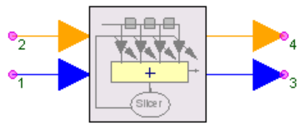
DFE Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
DFE (algorithm)	Decision-Feedback Equalizer
BlindDFE (algorithm)	Blind Decision Feedback Equalizer

DFE



Description: Decision-Feedback Equalizer
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *DFE Part* (algorithm), *BlindDFE Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
ChannelImpulse	Channel impulse response vector representing the overall impulse response from bit source to the input of FFE	[1]		None	NO
SamplesPerBit		16		Integer	NO
NumDFETaps	Number of DFE taps	0		Integer	NO
SampleInterval	Sample interval associated with the channel impulse response	1e-10	s	Float	NO
LogicLevel	Logic level of the binary bits injected into the channel. Bit 1 and 0 are represented by LogicLevel and -LogicLevel.	1.0	V	Float	NO
BitSampling	Bit sampling point: Negative clock edge, Positive clock edge	Positive clock edge		Enumeration	NO
AdaptiveEQ	Adaptive equalization option: NO, YES	YES		Enumeration	NO
Alpha	Taps update factor	0.01		Float	NO
StartThreshold	Magnitude threshold of the input signal to start the adaptive process	0		Float	NO
PrecedingEqualizer	Equalizer preceded before DFE: None, FFE	None		Enumeration	NO
FFENumPrecursorTaps	Number of FFE precursor taps. Total FFE taps = NumPrecursorTaps + NumPostcursorTaps + 1.	0		Integer	NO
FFENumPostcursorTaps	Number of FFE postcursor taps. Total FFE taps = NumPrecursorTaps + NumPostcursorTaps + 1.	0		Integer	NO
FFENormalization	Normalize FFE coefficients to have unit sum: NO, YES	NO		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO
2	clock	int	NO

Output Ports

Port	Name	Signal Type	Optional
3	output	real	NO
4	bit	int	NO

Notes/Equations

1. DFE implements adaptive decision feedback equalizer and automatically computes optimal bit-level DFE taps based on the given channel impulse response (parameter **ChannelImpulse**).
2. This model consumes one sample from the **input** port and one sample from the **clock** port and produces one sample to the **output** port and one sample to the **bit** port in every execution.
3. Parameter **PrecedingEqualizer** specifies the equalizer precedes DFE. If **PrecedingEqualizer** is set to *None*, parameter **ChannelImpulse** specifies the overall impulse response of the system starting right after the bit source and stopping right before the DFE. If **PrecedingEqualizer** is set to *FFE*, parameter **ChannelImpulse** specifies the overall impulse response of the system starting right after the bit source and stopping right before the *FFE* (algorithm). When computing optimal DFE taps, the effect of *FFE* (algorithm) will be taken into account.
4. Parameter **NumDFETaps** specifies the number of desired DFE taps.
5. Please refer to *FFE* (algorithm) for parameters **ChannelImpulse**, **SamplesPerBit**, **SampleInterval**, and **LogicLevel**.
6. Please refer to *BlindDFE* (algorithm) for parameters **BitSampling**, **AdaptiveEQ**, **Alpha**, and **StartThreshold**.
7. If **PrecedingEqualizer** is set to *FFE*, please refer to *FFE* (algorithm) for parameters

FFENumPrecursorTaps, **FFENumPostcursorTaps**, and **FFENormalization**.

8. Please refer to *BlindDFE* (algorithm) for DFE operation.
9. Use *BlindDFE* (algorithm) instead of *DFE* (algorithm) if you want to specify a set of custom taps.

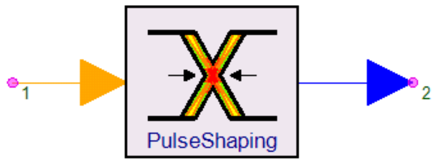
JitterGenerator Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
JitterGenerator (algorithm)	Jitter generator

JitterGenerator



Description: Jitter generator
Domain: Timed
C++ Code Generation Support: YES
Associated Parts: *JitterGenerator Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
VHigh	the Logic-1 voltage level	1	V	Float	NO
VLow	the Logic-0 voltage level	-1	V	Float	NO
EdgeTime	the rise and fall times (in second) : must be in the range between 0.0 and 0.5 UI	0.0	s	Float	NO
SamplesPerUI	the number of samples per UI	16		Integer	NO
InitialState	the initial state for the sequence	0		Integer	NO
DCD	Duty-cycle distortion (in second): must be in the range between 0.0 and 1.0 UI	0	s	Float	YES
PJ_Amplitude	Periodic jitter amplitude in sec	0	s	Float	YES
PJ_Frequency	Periodic jitter frequency in Hz	0	Hz	Float	YES
RJ	RMS random jitter (in second):must be in the range between 0.0 and 1.0 UI	0	s	Float	YES

Input Ports

Port	Name	Description	Signal Type	Optional
1	Input	the input sequences	int	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Output	the output sampled bits	real	NO

Notes/Equations

1. This model is used to apply random, periodic and DCD jitter to the input bit. Upsampling with rise/fall edge is also applied.
2. Each firing,
   - 1 token is consumed at the input port.
   - SamplesPerUI tokens are produced at the output port. It equals to 16 by default.
3. Parameter details:
   - VHigh specifies the logic-1 voltage level of the input sequences.
   - VLow specifies the logic-0 voltage level of the input sequences.
   - EdgeTime specifies the rise/fall times in second.
   - SamplesPerUI specifies the number of samples for one input bit.
   - InitialState specifies the initial state of the input sequence.
   - DCD specifies the Duty-cycle distortion in second.
   - PJ_Amplitude specifies the periodic Jitter amplitude in second.
   - PJ_Frequency specifies the periodic Jitter frequency in Hz.
   - RJ specifies the random Jitter in second.
4. The input sequences should be logical bit which can only be 1 or 0. This is a timed model so that bit rate can be got from the input port. Timing Jitter is defined as the deviation of a signal transition from its ideal position in time. The total Jitter consists of two components: the Deterministic Jitter(DJ) and Random Jitter(RJ).
   - DCD is a part of Deterministic Jitter which is caused by a difference in propagation delay between low to high transitions and high to low transitions. Assuming the ideal signal has 50% duty cycle, the deviated signal with DCD has non-50% duty cycle.

- Periodic Jitter is a part of Deterministic Jitter which is caused by electromagnetic interference. This type of deviation is referred to as Sinusoidal which repeats in a cyclic fashion. The model is represented as: $PJ_{total}(t) = \sum_{i=0}^{N} A_i \cos(\omega_i t + \theta_i)$ where $PJ_{Total}(t)$ denotes the total periodic jitter, N is the number of cosine components(tones), Ai is the amplitude in units of time, ω_i is the modulation frequency, t is the time and θ_i is the initial phase. This model is applied as:
  $PJ_{total}(t) = PJ\_Amplitude * \cos(2\pi * PJ\_Frequency * t)$.
- Random Jitter which comes from device noise sources such as thermal and flicker noise is characterized by both Gaussian and non-Gaussian distributions. This model only support Gaussian distribution with mean value equals to 0 and standard deviation equals to 1.

5. This model causes one bit time delay.

References

1. Kyung Ki Kim, Jing Huang, Yong-Bin Kim, Fabrizio Lombard. "On the Modeling and Analysis of Jitter in ATE Using Matlab". Proceedings of the 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 285-293,2005.

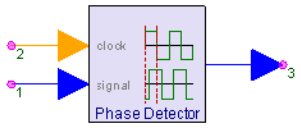
PhaseDetector Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
PhaseDetector (algorithm)	PhaseDetector

PhaseDetector



Description: PhaseDetector
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *PhaseDetector Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
BitCenter	Bit center and clock edge alignment: Negative clock edge, Positive clock edge	Positive clock edge		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	signal	real	NO
2	clock	int	NO

Output Ports

Port	Name	Signal Type	Optional
3	phaseError	real	NO

Notes/Equations

1. PhaseDetector detects phase difference (phase error) between input **signal** and reference **clock**.
2. The phase detection method used in this model is derived from: C. R. Hogge Jr. "A self correcting clock recovery circuit", IEEE Transactions on Electron Devices, 1985.
3. This model consumes one sample from **signal** and one sample from **clock** and produces one sample to **phaseError** in every execution.
4. PhaseDetector assumes **signal** is in NRZ (non-return-to-zero) format. If a signal sample is larger than 0, it is treated as bit level 1, otherwise, it is treated as bit level 0.
5. PhaseDetector treats positive-valued clock sample as clock level 1 and treats non-positive-valued clock sample as clock level 0.
6. The PhaseDetector detects phase difference (phase error) between the bit transition of the input signal and the clock edge.
7. Suppose **BitCenter** is set to *Negative clock edge*, **signal** and **clock** are synchronized (no phase error) if the negative clock edge is aligned with the bit center. On the other hand, suppose **BitCenter** is set to *Positive clock edge*, there is no phase error if the positive clock edge is aligned with the bit center.
8. PhaseDetector can only detect phase error when there is a bit transition in the input signal. When there is a bit transition, the phase error is outputted at the next positive clock edge if **BitCenter** is set to *Negative clock edge*. Otherwise (**BitCenter** is set to *Positive clock edge*), the phase error is outputted at the next negative clock edge. For all other situations, **phaseError** outputs 0.
9. Phase error is computed in rad. However, due to discrete-time implementation, the number of possible phase error values are equivalent to the number of samples per bit (samples per unit interval), which are equally spaced in the range of $\{-\pi, \pi\}$.

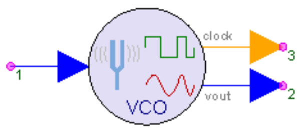
VCO Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
VCO (algorithm)	Voltage Controlled Oscillator

VCO



Description: Voltage Controlled Oscillator
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *VCO Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
SampleInterval	Sample interval in sec.	1e-10	s	Float	NO
Frequency	Fundamental oscillator frequency in Hz.	1e9	Hz	Float	NO
Sensitivity	Sensitivity in Hz/V.	1		Float	NO
InitialPhase	Initial phase in rad.	0	deg	Float	NO
Amplitude	Output amplitude in volt.	1	V	Float	NO

Input Ports

Port	Name	Signal Type	Optional
1	vin	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	vout	real	NO
3	clock	int	NO

Notes/Equations

1. VCO implements voltage-controlled oscillator.
2. This model consumes one sample from **vin** and produces one sample to **vout** and to **clock** in every execution.
3. The continuous-time input-output relationship of voltage-controlled oscillator can be expressed as $v_{out}(t) = A\ cos(2\pi f_0 t + 2\pi\alpha\int_0^t v_{in}(\tau)d\tau + \theta)$, where v_{in} represents input voltage **vin**, v_{out} represents output voltage **vout**, f_0 represents oscillator fundamental **Frequency** in Hz, α represents **Sensitivity** in Hz/volt, θ represents **InitialPhase** in rad, and A represents **Amplitude** in volt.
4. VCO is implemented as discrete-time version of the above equation. The sampling time instances are at $n\Delta t$, where $n = 0, 1, 2, ...$ and Δt represents sampling **TimeStep**.
5. The **clock** port outputs square clock waveform samples. The clock samples are generated by slicing v_{out}

   using the following equation:

$$clock(n\Delta t) = \begin{cases} 1, & v_{out}(n\Delta t) > 0\ , \\ 0, & \text{otherwise.} \end{cases}$$

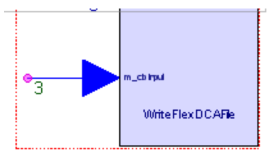
WriteFlexDCAFile Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
WriteFlexDCAFile (algorithm)	Write samples in a text file that conforms to the Agilent FlexDCA Software requirements

WriteFlexDCAFile



Description: Write samples in a text file that conforms to the Agilent FlexDCA Software requirements
Domain: Timed
C++ Code Generation Support: YES
Associated Parts: *WriteFlexDCAFile Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
FileName	Wave samples text file	MyPatternWave.txt		Filename	NO
NumToSkip	Number of bits to skip before writing to text file.	0		Integer	NO
NumToCollect	Number of bits to be collected and written into the text file.	1		Integer	NO
PeriodicPattern	Bits pattern is periodic (Required for jitter analysis): NO, YES	YES		Enumeration	NO
BitRate	Original bit rate.	Sample_Rate	Hz	Float	NO
PatternLength	Number of bits per period for periodic bit pattern.	1		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	m_cbInput	real	NO

Notes/Equations

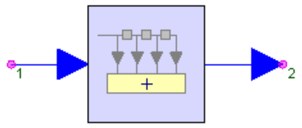
BlindFFE Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
BlindFFE (algorithm)	Blind Feed-Forward Equalizer
FFE (algorithm)	Feed-Forward Equalizer

BlindFFE



Description: Blind Feed-Forward Equalizer
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *FFE Part* (algorithm), *BlindFFE Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Coefficients	Bit level FFE taps	[1]		None	NO
SamplesPerBit	Number of samples per bit	16		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. BlindFFE implements feed-forward equalizer based on the given FFE taps (parameter **Coefficients**).
2. This model consumes one sample from the **input** port and produces one sample to the **output** port.
3. It is called "blind" because no training sequence is used to compute the taps.
4. Parameter **Coefficients** specifies the FFE taps. The taps are assumed at **bit level** to remove inter-symbol interference (ISI).
5. Parameter **SamplesPerBit** specifies the number of samples per bit used in the simulation, which is equivalent to bit time (unit interval (UI)) divided by sample interval.
6. Let $x[n]$

   and *y[n]* denote input sample sequence and output sample sequence respectively. Let

   $w_0, w_1, \ldots, w_{M-1}$

   denote FFE taps, where *M* is the number of taps. Let *N* denote number of samples per bit. The FFE operation can be expressed as:

   $y[n] = \sum_{i=0}^{M-1} w_i \; x[n - iN].$
7. See also *FFE* (algorithm). *FFE* (algorithm) automatically computes optimal taps based on the given channel impulse response.

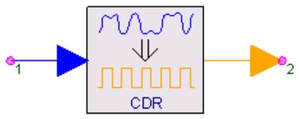
CDR Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
CDR (algorithm)	Clock and Data Recovery

CDR



Description: Clock and Data Recovery
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *CDR Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
SamplesPerBit	Number of samples per bit	16		Integer	NO
ZeroCrossing	Clock edge at zero crossing point: Negative clock edge, Positive clock edge	Negative clock edge		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	clock	int	NO

Notes/Equations

1. The CDR (clock and data recovery) model implements a very simple clock recovery mechanism based on zero crossing detection.
2. This model is mainly served as a C++ template for users to create custom CDR model.
3. For practical clock recovery, SystemVue provides *PhaseDetector* (algorithm) and *VCO* (algorithm) models to construct a PLL (phase locked loop) based clock recovery system.
4. This model consumes one sample from the **input** port and produces one sample to the **clock** port in every execution.
5. Parameter **SamplesPerBit** specifies the number of samples per bit used in the simulation, which is equivalent to bit time (unit interval (UI)) divided by sample interval.
6. This model detects zero crossing in the input signal, and uses the zero crossing point as positive clock edge (In other words, if locked, negative clock edge is aligned with bit center). If there is no zero-crossing, the clock period is maintained at **SamplesPerBit** samples.

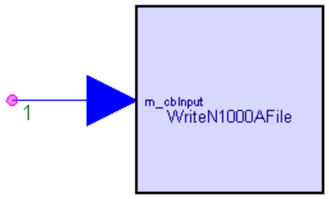
WriteN1000AFile Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
WriteN1000AFile (algorithm)	Write samples in a text file that conforms to the Agilent N1000A Software requirements

WriteN1000AFile



Description: Write samples in a text file that conforms to the Agilent N1000A Software requirements
Domain: Timed
C++ Code Generation Support: YES
Associated Parts: *WriteN1000AFile Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
FileName	Wave samples text file	MyPatternWave.txt		Filename	NO
NumToSkip	Number of bits to skip before writing to text file.	0		Integer	NO
NumToCollect	Number of bits to be collected and written into the text file.	1		Integer	NO
PeriodicPattern	Bits pattern is periodic (Required for jitter analysis): NO, YES	YES		Enumeration	NO
BitRate	Original bit rate.	Sample_Rate	Hz	Float	NO
PatternLength	Number of bits per period for periodic bit pattern.	1		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	m_cbInput	real	NO

Notes/Equations

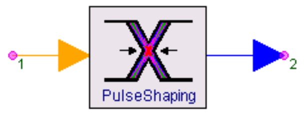
PulseShaping Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
PulseShaping (algorithm)	Jitter generator

PulseShaping



Description: Jitter generator
Domain: Timed
C++ Code Generation Support: YES
Associated Parts: *PulseShaping Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
VHigh	the Logic-1 voltage level	1	V	Float	NO
VLow	the Logic-0 voltage level	-1	V	Float	NO
EdgeTime	the rise and fall times (in second) : must be in the range between 0.0 and 0.5 UI	0.0	s	Float	NO
SamplesPerUI	the number of samples per UI	16		Integer	NO
InitialState	the initial state for the sequence	0		Integer	NO
DCD	Duty-cycle distortion (in second): must be in the range between 0.0 and 1.0 UI	0	s	Float	YES
PJ_Amplitude	Periodic jitter amplitude in sec	0	s	Float	YES
PJ_Frequency	Periodic jitter frequency in Hz	0	Hz	Float	YES
RJ	RMS random jitter (in second):must be in the range between 0.0 and 1.0 UI	0	s	Float	YES

Input Ports

Port	Name	Description	Signal Type	Optional
1	Input	the input sequences	int	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Output	the output sampled bits	real	NO

Notes/Equations

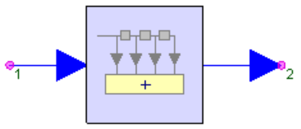
FFE Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
FFE (algorithm)	Feed-Forward Equalizer
BlindFFE (algorithm)	Blind Feed-Forward Equalizer

FFE



Description: Feed-Forward Equalizer
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *FFE Part* (algorithm), *BlindFFE Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
ChannelImpulse	Channel impulse response vector representing the overall impulse response from bit source to the input of FFE	[1]		None	NO
SamplesPerBit	Number of samples per bit	16		Integer	NO
NumPrecursorTaps	Number of precursor taps. Total FFE taps = NumPrecursorTaps + NumPostcursorTaps + 1.	0		Integer	NO
NumPostcursorTaps	Number of postcursor taps. Total FFE taps = NumPrecursorTaps + NumPostcursorTaps + 1.	0		Integer	NO
SampleInterval	Sample interval associated with the channel impulse response	1e-10	s	Float	NO
LogicLevel	Logic level of the binary bits injected into the channel. Bit 1 and 0 are represented by LogicLevel and -LogicLevel.	1.0	V	Float	NO
Normalization	Normalize FFE coefficients to have unit sum: NO, YES	NO		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. FFE implements feed-forward equalizer and automatically computes optimal bit-level FFE taps based on the given channel impulse response (parameter **ChannelImpulse**).
   This model consumes one sample from the **input** port and produces one sample to the **output** port.
2. Parameter **ChannelImpulse** specifies the overall impulse response of the system starting right after the bit source and stopping right before the FFE, which can possibly include transmitter, channel, and receiver frond-end, etc.
   - Let $\mathrm{h}[0], \mathrm{h}[1], \ldots, \mathrm{h}[\mathrm{L}\text{-}1]$

     denote such impulse response specified by **ChannelImpulse**, where *L* is the length of the given channel impulse response.
3. Parameter **SamplesPerBit** specifies the number of samples per bit used in the simulation, which is equivalent to bit time (unit interval (UI)) divided by sample interval.
4. Parameter **NumPrecursorTaps** and **NumPostcursorTaps** specify the number of precursor taps *N1* and the number of postcursor taps *N2* to be used in FFE. The total number of FFE taps including the main cursor is *N1+N2+1*.
5. Parameter **LogicLevel** specifies logic one voltage, *LogicLevel*, and logic zero voltage, -*LogicLevel*, in the NRZ (non-return-to-zero) format. The bit stream injected into the channel is assumed in the NRZ format.
   - Let *b[n]* = *1* or *0* represent random bits.
   - Let *N* denote number of samples per bit.
   - Set *V* = *LogicLevel*.
   - The sampled version of bits *b[k]* in the NRZ format can be expressed in *z[n]* as:
     $$z[kN], z[kN+1], \ldots, z[kN+N-1] = \begin{cases} V, & \text{if } b[k] = 1 \\ -V, & \text{if } b[k] = 0 \end{cases}$$
6. Parameter **SampleInterval** specifies sampling interval $\triangle t$

   used in the simulation. Sample interval is equivalent to 1 / sampling rate.
7. The FFE input signal *x[n]* is assumed to be the output of channel *h* when the input to the channel is *z[n]*. Assuming channel *h* is a LTI (linear time-invariant) system, then *x[n]* can be expressed as $x[n] = \sum_{i=0}^{L-1} h[i] z[n-i] \triangle t$

- The convolution in this context involves a scaling factor of sample interval. This convention is commonly adopted by many channel simulators.
- **ChannelImpulse** *h[n]* should be scaled properly to take sample interval into account.

8. Based on the given channel impulse and other parameters, FFE automatically computes a set of bit-level taps $w_0, w_1, \ldots, w_{M-1}$ to remove ISI (inter symbol interference). The number of FFE taps *M* is *N1_N2_1*.
9. When parameter **Normalization** is set to *NO*, the amplitude of output samples is approximately in the range of 1 and -1. When parameter **Normalization** is set to *YES*, then *w[n]* is normalized to have unit sum, i.e., $\sum_{i=0}^{M-1} w_i = 1$
10. Let *y[n]* denote the output samples of FFE. The FFE operation can be expressed as: $y[n] = \sum_{i=0}^{M-1} w_i \; x[n - iN].$
11. See also *BlindFFE* (algorithm). Use *BlindFFE* (algorithm) instead of FFE if you want to specify a set of custom taps.

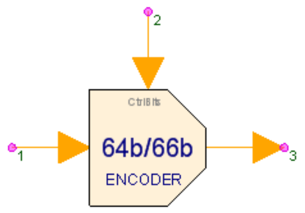
Coder64b66b Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
Coder64b66b (algorithm)	64b/66b encoder

Coder64b66b



Description: 64b/66b encoder
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Coder64b66b Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Scrambler	scramble or not: NO, YES	NO		Enumeration	NO
ScramblerInit	initial state of scrambler	[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]		Integer array	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	int	NO
2	CtrlBits	int	NO

Output Ports

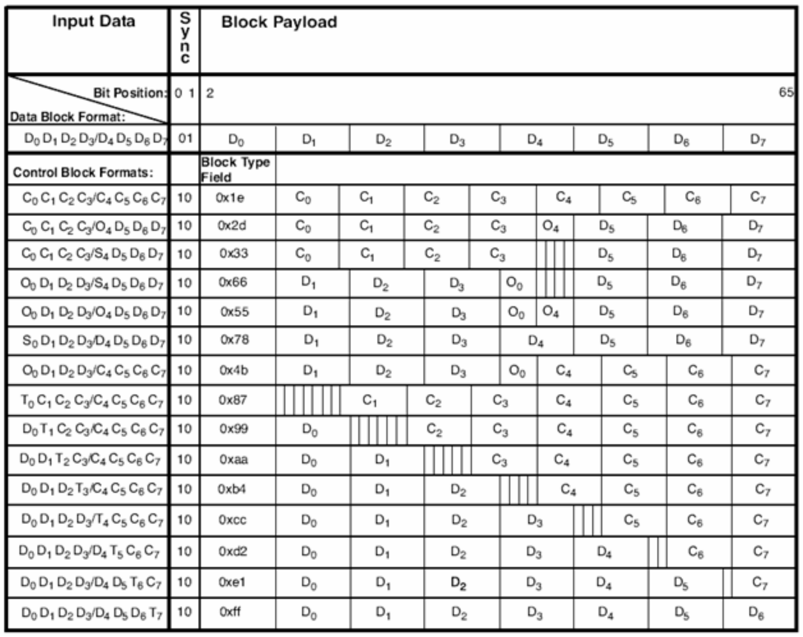
Port	Name	Signal Type	Optional
3	output	int	NO

Notes/Equations

1. The 64B/66B transmission code is used to improve the transmission characteristics of information and to support transmission of control and data characters. The encodings defined by the transmission code ensure that sufficient transitions are present in the PHY bit stream to make clock recovery possible at the receiver. Such encoding also greatly increases the likelihood of detecting any single or multiple bit errors that may occur during transmission and reception of information. In addition, the synchronization headers of the code enable the receiver to achieve block alignment on the incoming PHY bit stream. The 64B/66B transmission code has a high transition density and is a run-length-limited code.
2. 64B/66B encodes 8 data octets or control characters into a block. Blocks containing control characters also contain a block type field. Data octets are labeled D $_0$ to D $_7$. Control characters other than /O/, /S/ and /T/ are labeled C0 to C $_7$. The control character for ordered_set is labeled as O $_0$ or O $_4$ since it is only valid on the first octet of the XGMII. The control character for start is labeled as S $_0$ or S $_4$ for the same reason. The control character for terminate is labeled as T $_0$ to T $_7$.
   Two consecutive XGMII transfers provide eight characters that are encoded into one 66-bit transmission block. The subscript in the above labels indicates the position of the character in the eight characters from the XGMII transfers.
   Contents of block type fields, data octets and control characters are shown as hexadecimal values. The LSB of the hexadecimal value represents the first transmitted bit. For instance, the block type field 0x1e is sent from left to right as 01111000. The bits of a transmitted or received block are labeled TxB<65:0> and RxB<65:0> respectively where TxB<0> and RxB<0> represent the first transmitted bit. The value of the sync header is shown as a binary value. Binary values are shown with the first transmitted bit (the LSB) on the left.
3. Blocks consist of 66 bits. The first two bits of a block are the synchronization header (sync header). Blocks are either data blocks or control blocks. The sync header is 01 for data blocks and 10 for control blocks. Thus, there is always a transition between the first two bits of a block. The remainder of the block contains the payload. The payload is scrambled and the sync header bypasses the scrambler. Therefore, the sync header is the only position in the block that always contains a transition. This feature of the code is used to obtain block synchronization.
   Data blocks contain eight data characters. Control blocks begin with an 8-bit block type field that indicates the format of the remainder of the block. For control blocks containing a Start or Terminate character, that character is implied by the block type field. Other control characters are encoded in a 7-bit control code or a 4-bit O Code. Each control block contains eight characters.

The format of the blocks is as shown in 64B/66B Encoder. In the figure, the column labeled Input Data shows, in abbreviated form, the eight characters used to create the 66-bit block. These characters are either data characters or control characters and, when transferred across the XGMII interface, the corresponding TXC or RXC bit is set accordingly. Within the Input Data column, D0 through D7 are data octets and are transferred with the corresponding TXC or RXC bit set to zero. All other characters are control octets and are transferred with the corresponding TXC or RXC bit set to one. The single bit fields (thin rectangles with no label in the figure) are sent as zero and ignored upon receipt.
Bits and field positions are shown with the least significant bit on the left. Hexadecimal numbers are shown in normal hexadecimal. For example the block type field 0x1e is sent as 01111000 representing bits 2 through 9 of the 66 bit block. The least significant bit for each field is placed in the lowest numbered position of the field.

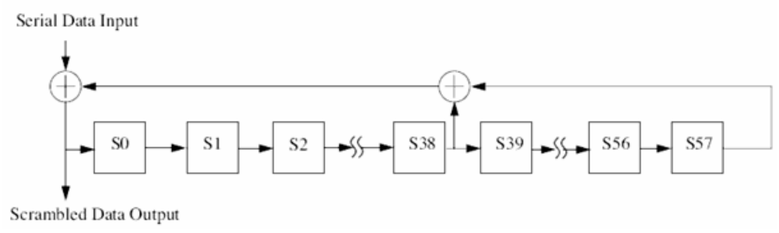
64B/66B Encoder



Input Data	Sync	Block Payload							
Bit Position: / Data Block Format:	0 1	2							65
$D_0 D_1 D_2 D_3/D_4 D_5 D_6 D_7$	01	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
Control Block Formats:		Block Type Field							
$C_0 C_1 C_2 C_3/C_4 C_5 C_6 C_7$	10	0x1e	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7
$C_0 C_1 C_2 C_3/O_4 D_5 D_6 D_7$	10	0x2d	C_0	C_1	C_2	C_3	O_4	D_5	D_6	D_7
$C_0 C_1 C_2 C_3/S_4 D_5 D_6 D_7$	10	0x33	C_0	C_1	C_2	C_3		D_5	D_6	D_7
$O_0 D_1 D_2 D_3/S_4 D_5 D_6 D_7$	10	0x66	D_1	D_2	D_3	O_0		D_5	D_6	D_7
$O_0 D_1 D_2 D_3/O_4 D_5 D_6 D_7$	10	0x55	D_1	D_2	D_3	O_0	O_4	D_5	D_6	D_7
$S_0 D_1 D_2 D_3/D_4 D_5 D_6 D_7$	10	0x78	D_1	D_2	D_3	D_4	D_5	D_6	D_7	
$O_0 D_1 D_2 D_3/C_4 C_5 C_6 C_7$	10	0x4b	D_1	D_2	D_3	O_0	C_4	C_5	C_6	C_7
$T_0 C_1 C_2 C_3/C_4 C_5 C_6 C_7$	10	0x87		C_1	C_2	C_3	C_4	C_5	C_6	C_7
$D_0 T_1 C_2 C_3/C_4 C_5 C_6 C_7$	10	0x99	D_0		C_2	C_3	C_4	C_5	C_6	C_7
$D_0 D_1 T_2 C_3/C_4 C_5 C_6 C_7$	10	0xaa	D_0	D_1		C_3	C_4	C_5	C_6	C_7
$D_0 D_1 D_2 T_3/C_4 C_5 C_6 C_7$	10	0xb4	D_0	D_1	D_2		C_4	C_5	C_6	C_7
$D_0 D_1 D_2 D_3/T_4 C_5 C_6 C_7$	10	0xcc	D_0	D_1	D_2	D_3		C_5	C_6	C_7
$D_0 D_1 D_2 D_3/D_4 T_5 C_6 C_7$	10	0xd2	D_0	D_1	D_2	D_3	D_4		C_6	C_7
$D_0 D_1 D_2 D_3/D_4 D_5 T_6 C_7$	10	0xe1	D_0	D_1	D_2	D_3	D_4	D_5		C_7
$D_0 D_1 D_2 D_3/D_4 D_5 D_6 T_7$	10	0xff	D_0	D_1	D_2	D_3	D_4	D_5	D_6	

4. Ordered sets are used to extend the ability to send control and status information over the link such as remote fault and local fault status. Ordered sets consist of a control character followed by three data characters. Ordered sets always begin on the first octet of the XGMII. 10 Gigabit Ethernet uses one kind of ordered_set: the sequence ordered_set. The sequence ordered_set control character is denoted /Q/. An additional ordered_set, the signal ordered_set, has been reserved and it begins with another control code. The 4-bit O field encodes the control code. See Table 49-1 in IEEE Std 802.3ae-2002, Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, Amendment: Media Access Control (MAC) Parameters, Physical Layers, and Management Parameters for 10 Gb/s Operation, Section 49.2. for the mappings.
5. A block is invalid if any of the following conditions exists:
a) The sync field has a value of 00 or 11.
b) The block type field contains a reserved value.
c) Any control character contains a value not in Table 49-1.
d) Any O code contains a value not in Table 49-1.
e) The set of eight XGMII characters does not have a corresponding block format in 64B/66B Encoder.
6. If parameter *Scrambler* is set as NO, the payload of the block is not scrambled. If it is set as *YES*, the payload of the block is scrambled with a self-synchronizing scrambler. The scrambler shall produce the same result as the implementation shown in Scrambler. This implements the scrambler polynomial: G(x) = 1 + x39 + x58. The parameter *ScramblerInit* is the initial value of the scrambler according to Scrambler. Note that, in this 58-element array parameter *ScramblerInit* , the first element is the initial value in S0 while the 58th element is the initial value in S57. The scrambler is run continuously on all payload bits. The sync header bits bypass the scrambler.

Scrambler



7. Each firing,
    - 64 tokens are consumed at pin input, and 8 tokens are consumed at pin CtrlBits. 66 tokens are produced at pin output.
    - The input at pin input are 8 data octets or control characters. For each data octet or control character, the LSB is input first.
    - Each token at pin CtrlBits indicates the type of corresponding octet at pin input. 0 indicates data octet while 1 indicates control character.
    - All the bits are input and output serially.

References

1. IEEE Std 802.3ae-2002, Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, Amendment: Media Access Control (MAC) Parameters, Physical Layers, and Management Parameters for 10 Gb/s Operation, Section 49.2.

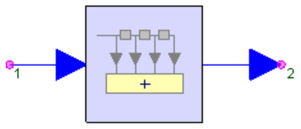
TimeResponseFIR Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model
TimeResponseFIR (algorithm)

TimeResponseFIR



Description:
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *TimeResponseFIR Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
ResponseType	Type of Response specified: Impulse Response, Step Response	Step Response		Enumeration	NO
Response	Time Response Coefficients			Floating point array	NO
ResponseTimeStamps	Time Stamps of time response coefficients, must be in ascending order			Floating point array	NO
SampleRate	Input signal sampling rate	10.3125e9*16	Hz	Float	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

TimeResponseFIR can be used to recharacterize step or impulse response specified with **Response** and **ResponseTimeStamps** at a user specified **TimeStep**. The user specefied **Response** is usually measured either in the lab using measurement equipments or using circuit simulators such as H-Spice. If the **ResponseType** is **Impulse Response** then a 4-point Lagrange interpolation is performed to calculate recharacterized impulse response with specified Time Step. If the **ResponseType** is **Step Response** then analytical differentiation is performed on input response in the time vicinity of each coefficient of recharacterized impulse response to calculate the value of the coefficient at the accurate time specified using Time Step. The recharacterized impulse response coefficients are then used as FIR taps. In case of **ResponseType** being Step Response, output of the FIR is also multiplied with **TimeStep** value.

The number of coefficients in the recharacterized impulse response are calculated as (max (ResponseTimeStamps) - min (ResponseTimeStamps)) / TimeStep.

The values in ResponseTimeStamps must be positive and in non-repeating ascending order.

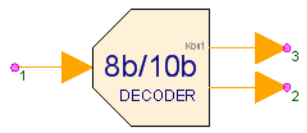
Decoder8b10b Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
Decoder8b10b (algorithm)	8b/10b decoder

Decoder8b10b



Description: 8b/10b decoder
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Decoder8b10b Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Delay	number of 10-bit symbol delay	0		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	int	NO

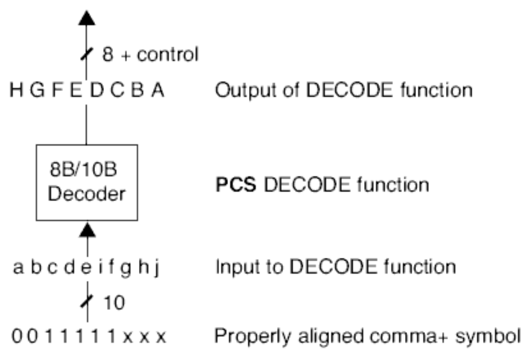
Output Ports

Port	Name	Signal Type	Optional
2	output	int	NO
3	Kout	int	NO

Notes/Equations

1. The 8B/10B decoder is the reverse procedure of 8B/10B encoder. It's illustrated in Decoder 8B/10B.

   Decoder 8B/10B

   For more information on the 8B/10B Coder, refer to *Coder8b10b* (algorithm).
2. Parameter Description:
   Delay specifies the number of 10-bit symbol delay. The decoder begins to work after 10*Delay input tokens.
3. Each firing,
   - Ten tokens are consumed at pin input. One token is produced at pin Kout (control character), and eight tokens are produced at pin output.
   - All the bits are input and output serially.
   - The input at pin input is the 10-bit transmission code-group. The LSB bit (a) is input first, while the MSB (j) is input last.
   - The output at pin Kout is the decoded control variable Z, in which 0 means the value D and 1 means the value K.
   - The output at pin output is the decoded information octet. The LSB bit (A) is output first, while the MSB (H) is output last.

References

1. IEEE Std 802.3, 2000 Edition, Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, Section 36.2.4.

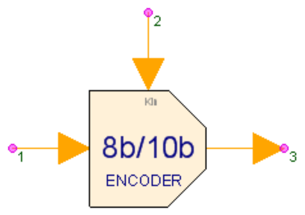
Coder8b10b Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
Coder8b10b (algorithm)	8b/10b encoder

Coder8b10b



Description: 8b/10b encoder
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Coder8b10b Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	Din	int	NO
2	Kin	int	NO

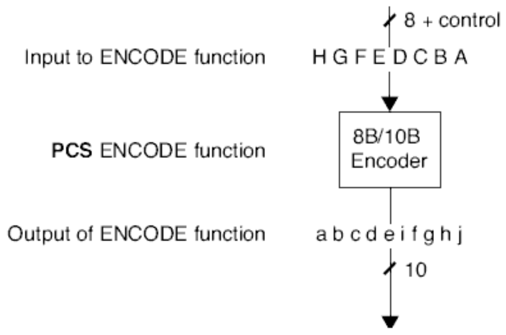
Output Ports

Port	Name	Signal Type	Optional
3	output	int	NO

Notes/Equations

1. The 8B/10B transmission code is used to improve the transmission characteristics of information. The encodings defined by the transmission code ensure that sufficient transitions are present in the PHY bit stream to make clock recovery possible at the receiver. Such encoding also greatly increases the likelihood of detecting any single or multiple bit errors that may occur during transmission and reception of information. In addition, some of the special code-groups of the transmission code contain a distinct and easily recognizable bit pattern that assists a receiver in achieving code-group alignment on the incoming PHY bit stream. The 8B/10B transmission code has a high transition density, is a run-length-limited code, and is dc-balanced. The transition density of the 8B/10B symbols ranges from 3 to 8 transitions per symbol.
2. 8B/10B transmission code uses letter notation for describing the bits of an unencoded information octet and a single control variable. Each bit of the unencoded information octet contains either a binary zero or a binary one. A control variable, Z, has either the value D or the value K. When the control variable associated with an unencoded information octet contains the value D, the associated encoded code-group is referred to as a data code-group. When the control variable associated with an unencoded information octet contains the value K, the associated encoded code-group is referred to as a special code-group.
   The bit notation of A, B, C, D, E, F, G, H for an unencoded information octet is used in the description of the 8B/10B transmission code. The bits A, B, C, D, E, F, G, H are translated to bits a, b, c, d, e, i, f, g, h, j of 10-bit transmission code-groups. The 8B/10B encoder is illustrated in Coder8b/10b. Each valid code-group has been given a name using the following convention: /Dx.y/ for the 256 valid data code-groups, and /Kx.y/ for special control code-groups, where x is the decimal value of bits EDCBA, and y is the decimal value of bits HGF. For detailed information, refer to *Tables 36-1* and *36-2* in *IEEE Std 802.3, 2000 Edition, Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, Section 36.2.4.*

8B/10B Encoder



3. Each firing,

- Eight tokens are consumed at pin *Din*, and one token is consumed at pin *Kin* (control character). Ten tokens are produced at pin output.
- All the bits are input and output serially.
- The input at pin *Kin* is the control variable Z, in which 0 means the value D and 1 means the value K.
- The input at pin *Din* is the unencoded information octet. The LSB bit (A) is input first, while the MSB (H) is input last.
- The output at pin output is the 10-bit transmission code-group. The LSB bit (a) is output first, while the MSB (j) is output last.

References

1. IEEE Std 802.3, 2000 Edition, Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, Section 36.2.4.

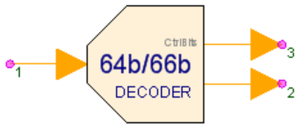
Decoder64b66b Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
Decoder64b66b (algorithm)	64b/66b decoder

Decoder64b66b



Description: 64b/66b decoder
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Decoder64b66b Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Scrambler	scramble or not: NO, YES	NO		Enumeration	NO
ScramblerInit	initial state of scrambler	[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]		Integer array	NO
Delay	number of 66-bit symbol delayed for descrambler	0		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	int	NO

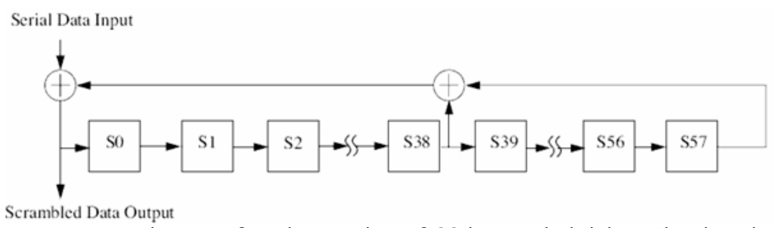
Output Ports

Port	Name	Signal Type	Optional
2	output	int	NO
3	CtrlBits	int	NO

Notes/Equations

1. The 64B/66B decoder is the reverse procedure of the 64B/66B encoder. For more information on the 64B/66B Coder, refer to *Coder64b66b* (algorithm).
2. Parameter Description:
   If parameter *Scrambler* is set as *NO*, the payload of the block is not scrambled. If it is set as *YES*, the payload of the block is scrambled with a self-synchronizing scrambler. The scrambler shall produce the same result as the implementation shown in Scrambler. This implements the scrambler polynomial: G(x) = 1 + x39 + x58. The parameter *ScramblerInit* is the initial value of the scrambler according to Scrambler. Note that, in this 58-element array parameter *ScramblerInit*, the first element is the initial value in S0 while the 58th element is the initial value in S57. The scrambler is run continuously on all payload bits. The sync header bits bypass the scrambler.

   Scrambler

   Parameter *Delay* specifies the number of 66-bit symbol delay. The decoder begins to work after 66* *Delay* input tokens.
3. Each firing,
   - 66 tokens are consumed at pin input. 64 tokens are produced at pin output, and 8 tokens are produced at pin CtrlBits (with each corresponding to 8 decoded bits).
   - The output at pin output are 8 data octets or control characters. For each data octet or control character, the LSB is input first.
   - Each token at pin CtrlBits indicates the type of corresponding output octet at pin output. 0 indicates data octet while 1 indicates control character.
   - All the bits are input and output serially.

References

1. IEEE Std 802.3ae-2002, Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, Amendment: Media Access Control (MAC) Parameters, Physical Layers, and Management Parameters for 10 Gb/s Operation, Section 49.2.

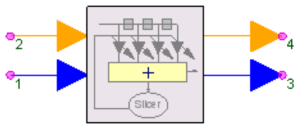
BlindDFE Part

Categories: *C++ Code Generation* (algorithm), *IBIS-AMI Transceivers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
BlindDFE (algorithm)	Blind Decision Feedback Equalizer
DFE (algorithm)	Decision-Feedback Equalizer

BlindDFE Part



Description: Blind Decision Feedback Equalizer
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *DFE Part* (algorithm), *BlindDFE Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Coefficients	Bit level DFE taps	[0]		None	NO
BitSampling	Bit sampling point: Negative clock edge, Positive clock edge	Positive clock edge		Enumeration	NO
AdaptiveEQ	Adaptive equalization option: NO, YES	YES		Enumeration	NO
Alpha	Taps update factor	0.01		Float	NO
ScalingFactor	Scaling equalized signal before slicer	1.0		Float	NO
StartThreshold	Magnitude threshold of the input signal to start the adaptive process	0		Float	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO
2	clock	int	NO

Output Ports

Port	Name	Signal Type	Optional
3	output	real	NO
4	bit	int	NO

Notes/Equations

1. BlindDFE implements adaptive decision feedback equalizer.
2. This model consumes one sample from the **input** port and one sample from the **clock** port and produces one sample to the **output** port and one sample to the **bit** port in every execution.
3. It is called "blind" because no training sequence is used to compute the taps.
4. Parameter **Coefficients** specifies a set of DFE taps $w_1, w_2, \ldots, w_M$

   .
5. Let *x[n]* represent the input samples and *y[n]* represent the output samples.
6. Let $b[k-1], b[k-2], \ldots, b[k-M]$

   represent the decision bits in the feedback delay line. In each execution of BlindDFE,

   $y[n] = x[n] + \sum_{i=1}^{M} w_i \; b[k-i]$

   .
7. Let *c[n]* represent the input clock samples. A positive clock edge occurs at sample instance *n* if *c[n] > 0* and *c[n-1] <= 0*. A negative clock edge occurs at sample instance *n* if *c[n] <= 0* and *c[n-1] > 0*.
8. Parameter **BitSampling** specifies clock edges for the decision instances and update-shift instances. When **BitSampling** is set to *Negative clock edge*, then decision instances are at negative clock edges and update-shift instances are at positive clock edges. When **BitSampling** is set to *Positive clock edge*, then decision instances are at positive clock edges and update-shift instances are at negative clock edges.
9. At decision instances, *y[n]* is scaled by parameter **ScalingFactor** (denoted as *C*) before sending to the slicer. The decision bit *b[k]* is computed as:

   $$b[k] = \begin{cases} 1, & \text{if } Cy[n] > 0 \\ -1, & \text{if } Cy[n] <= 0 \end{cases}$$

   The decision bits can be obtained from the **bit** output port.
10. DFE error is defined as $\epsilon = Cy[n] - b[k]$ at decision instances.
11. Parameter **AdaptiveEQ** specifies whether adaptive equalization is turned on. If parameter **AdaptiveEQ** is set to *YES*, taps are updated at update-shift instances based on the LMS adaptive method: $w_i = w_i - \alpha \; \epsilon \; b[k-i]$

    for

$i = 1, 2, \ldots, M$

, where

α

is specified by parameter **Alpha**. In this case, parameter **Coefficients** specifies only the initial tap values.

12. On the other hand, if parameter **AdaptiveEQ** is set to *NO*, DFE taps are fixed throughout simulation.
13. At update-shift instances, feedback delay line of the past decision bits are updated as: $b[k-i] = b[k-i+1]$ for $i = 1, 2, \ldots, M$.
14. When adaptive equalization is turned on, parameter **StartThreshold** specifies when to turned on the adaptive process based on the magnitude threshold for the input signal. At initial decision instances, if the magnitude of the input signal is less than the threshold, taps will not be updated. This prevents updating taps from false errors due to initial transient stage of the input signal. Once the magnitude of the input signal exceeds the threshold at a particular decision instance, the adaptive process is turned on afterward.
15. See also *DFE* (algorithm). *DFE* (algorithm) automatically computes optimal taps based on the given channel impulse response.

SignalDownloader_E4438C Part

Signal Downloader for Agilent ESG E4438C and MXG N5182A RF Signal Synthesizers.

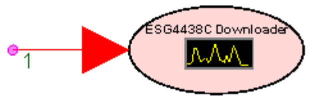
Categories: *Instrumentation* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model
SignalDownloader_E4438C (algorithm)

<\!\--start_panel_strip-\-><\!\--end_panel_strip-\->

SignalDownloader_E4438C



Description: SignalDownloader
Category: Instrumentation
Domain:
Associated Part: *SignalDownloader E4438C Part* (algorithm)

Basic Parameters

Name	Description	Default	Symbol	Unit	Type	Range
HWAvailable	YES: HW available. NO: HW not exist, communication to HW supressed	NO			enum	
IOType	Instrument Communication type: 'LAN', 'TCPIP', or 'GPIB'	LAN			string	
IOBoardID	For GPIB IO type only. Identify the IO board index. Typically 0 from default PC IO setup	0			string	
PrimAddress	Primary address: IP address (LAN) or instrument GPIB address (GPIB)	111.222.333.444			string	
SecAddress	Secondary address: Port number (LAN) or secondary address (GPIB)	5025			int	
TimeStart	Time to start waveform recording	Start_Time		second	float	[Start_Time,Stop_Time]
TimeStop	Time to stop waveform recording	Stop_Time		second	float	[Start_Time,Stop_Time]
FileName	File name for the downloaded waveform	esg.wfm			string	
RFPower	RF output power level in dBm	-35			dBm	
ArbOn	Turn ON/OFF ARB and mpdulation	OFF			enum	
RFPowerOn	Turn ON/OFF RF output	OFF			enum	
EventMarkers	Enable event marker pulse outputs: no output; Event1 output only; Event2 output only; or both Event1 and Event2 outputs	NONE			enum	
MarkerStart	Sample index to start event marker pulse (Note that sample index starts at 1)	1			int	[1,Num_Samples]
MarkerLength	Duration of event marker pulse measured in number of waveform samples	10			int	[1,Num_Samples]
ShowAdvancedParams	To display parameters for advanced instrument setups	NO			enum	

Advanced Parameters (after setting ShowAdvancedParams to YES)

Name	Description	Default	Symbol	Unit	Type	Range
Reset	Reset the instrument before everything else. YES: reset instrument; NO: skip instrument reset	YES			enum	
IQModFilter	Filter applied to played out IQ (ARB) waves: THROUGH (no filter applied); 2.1MHz Low Lass Filter; 40MHz Low pass Filter	THROUGH			enum	
SCPICommands	Additonal SCPI commands before turning ARB and RF output ON or OFF				string	
DownloadSize	Waveform download size. The number of samples to be streamed into the instrument per download operation	Num_Samples			int	[1,Num_Samples]
AutoScale	Normalize waveform into [-1, 1] Volt range before downloading. YES: auto scale; NO: no auto scale	YES			enum	
InstructionTimeout	Instrument instruction timeout limit	10		second	float	
DoDownload	Request to download waveform to instrument	YES			enum	
ARBRefSrc	Source for ARB reference frequency, either: provided internally by the instrument (INTERNAL) or provided by an external instrument (EXTERNAL)	INTERNAL			enum	
ARBRefFreq	ARB reference frequency (Active only when chosen EXTERNAL ARB reference)	10e6		Hz	float	
ConfigMIMO	Establish the Master-Slaves relationship among multiple E4438C or MXG(N5182A) signal generators. YES: Instrument participates MIMO setup; NO: Instrument doesn't participate MIMO setup	NO			enum	
MasterSrc	Mark this signal source as the master that will generate the synchronization trigger from EVENT1 (MXG) or EVENT2 (E4438C). YES: Master source; NO: Slave source	NO			enum	
UseE4438Cs	YES = E4438C's are used; NO = MXG's are used	YES			enum	
NumMXGSlaves	(For MXG's only) Number of MXG Slaves (range: 1 to 15)	1			int	[1, 15]
SlavePosition	(For slave MXG's only) Slave position (Range: 1 to 15)	1			int	[1, 15]

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	Complex Envelope (i.e. Modulated RF signal)	NO

Example: ESG SignalDownloader Example.wsv under *Instruments Examples* (examples)

⚠ **SignalDownloader_E4438C** supports both Agilent **ESG E4438C** and **MXG N5182A** instruments

⚠ **SignalDownloader_E4438C** **only** accepts **Timed Complex Envelope** (also known as **Timed modulated RF**) signal. Make sure you use a part such as ***RF Modulator* (algorithm)** part in front of it to convert complex IQ signal into **Complex Envelope** signal. If the complex IQ signal does not have sampling time information, make sure to use a ***SetSampleRate* (algorithm)** part in front of the ***RF Modulator* (algorithm)** part to set up the sampling rate for the baseband IQ waveform.

How to determine if a signal is "Timed"? Open the **Source** part (e.g. **SineGen**, **Bits**, etc)used in the design/schematic, set **"ShowAdvancedParams = YES"**; the **"SampleRateOption"** should be **"1:Timed from SampleRate" (recommended default setting)** or **"2:Timed from Schematic"** for timed signals.

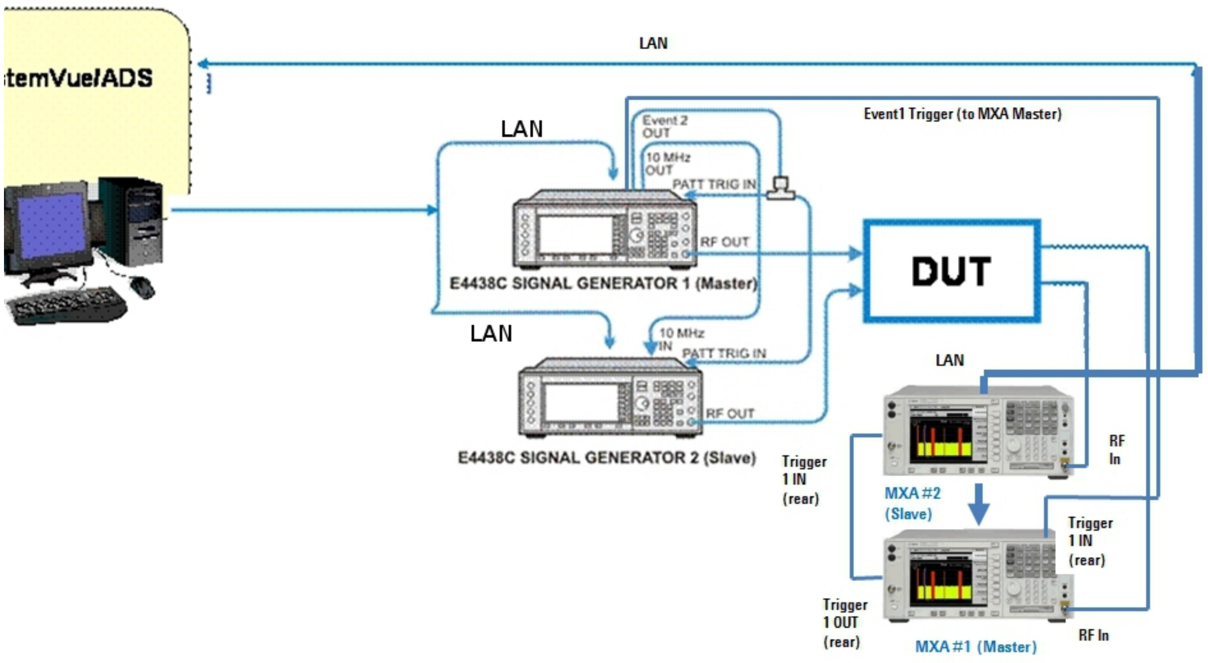
ℹ In current SystemVue release, "**LAN**" is the only **IOType** supported.

ℹ Where do the values for variables such as **Num_Samples**, **Start_Time** or **Stop_Time** come from? They all come from the simulation controller for Data Flow simulation.
Create a workspace from the **Blank** template. Then double click the "Design1 Analysis" on the workspace tree, it will bring up the "***Data Flow Analysis* (sim)**" window, the simulation controller.
The "**Start_Time**" contains the value in the **Start Time:** field, the "**Stop_Time**" for the **Stop Time:**, and of course, the "**Num_Samples**" for **Number of Samples**.

Master-Slave(s) Configuration for MIMO Applications

1. **Using E4438C's - Example: ESG MIMO Configuration Example.wsv under *Instruments Examples* (examples)**

- **Configure the interconnections between the master and slave E4438C's as shown in the diagram.** (Note: You can **add more slave E4438C's in the same way as the existing slave**).
- **The 10 MHz references should be connected between the instruments.**
- **The EVENT2 output of the master E4438C is used to generate the synchronization trigger and can't be used to as EventMarker outputs.**
- **Master or slave status is controlled by the MasterSrc parameter.**
- **Make sure to set ARBRefSrc = EXTERNAL**



1. **Using N5182A's - Example: MXG MIMO Configuration Example.wsv under *Instruments Examples* (examples)**

- **Make sure that the EVENT 1 output from the master MXG (N5182A) is connected to the PAT TRIG IN on the slave MXG.**
- **If additional slaves are present, the EVENT 1 output of each slave should be connected to the PAT TRIG IN of the next slave in the system.**
- **The 10 MHz references should be connected between the instruments.**
- **Make sure to leave ARBRefSrc at its default INTERNAL state. MXG's will automatically sense the reference signals and adjust itself accordingly.**
- **Make sure ArbOn is set to "OFF". You have to use *MathLang script* (users) to run the simulation followed by turning ON the baseband ARB.**
- **Make sure UseE4438Cs = "NO", and set up NumMXGSlaves for master MXG and set up NumMXGSlaves and SlavePosition for slave MXG('s) accordingly.**

Notes/Equations

1. **SignalDownloader_E4438C** supports both Agilent **ESG E4438C** and **MXG N5182A** instruments. It provides an interface for streaming baseband IQ waveforms modulated onto an RF carrier into the Agilent **E4438C** and **N5182A** Vector Signal Synthesizers to produce the desired modulated RF signals at the specified power level.
2. **SignalDownloader_E4438C only** accepts **Timed Complex Envelope** (also known as **Timed modulated RF**) signal. Make sure you use a part such as ***RF Modulator* (algorithm)** part in front of it to convert complex IQ signal into **Complex Envelope** signal. If the complex IQ signal does not have sampling time information, make sure to use a ***SetSampleRate* (algorithm)** part in front of the ***RF Modulator* (algorithm)** part to set up the sampling rate for the baseband IQ waveform.
3. The **EventMarkers** parameter controls whether a pulse will be sent out at the instrument's Event1 and Event2 output. The pulses coming out of the two outputs have the same characteristics. The pulse rises at the sample indexed by the **MarkerStart** parameter and last the number of samples determined by the **MarkerLength** parameter.
   For example, assume the waveform contains 600 samples (i.e. 600 IQ pairs), if you want a pulse to start at the 1st sample and last 10 sampling period, you simply set the **MarkerStart = 1** and **MarkerLength = 10**.
4. The **SCPICommands** parameter holds additional instrument control SCPI commands you want to send to the instrument. The SCPI commands entered here will be

executed right before the RF modulation and RF power are turned on. This means if you want to set RF powers or event markers again here, it will override the settings passed in through the **RFPower** or **EventMarkers** parameters.

5. The **DownloadSize** parameter determines the size of data package the simulator will send to the instrument during the waveform downloading operation. If the size is large, a sizable amount of memory will be required to hold the package before downloading it to the instrument. If the data package is small, there will be many downloading operations to download the complete waveform, which might slow down the downloading process.
   The default size **Num_Samples**, which is the total number of samples created by the simulation, works well if the it is less than 1M samples. If the total number of samples is larger than 1M samples, a size of 1M samples seems a good compromise between memory usage and downloading time since setting it larger than 1M provides no observable downloading time reduction.
6. The **AutoScale** parameter will force a normalization of the waveform generate to within [-1, 1]V range. The scaling is linear and the maximum magnitude of the I and Q data is scaled to -1V or 1V.
7. If an instruction can not be completed within the time specified in the **InstructionTimeout** parameter, either due to IO communication issues or due to hardware failures, a timeout will occur so that the simulator will not appear to hang.
8. Set **DoDownload = NO** if you want to run a simulation without downloading the waveform into the instrument.
9. When you choose to hook your instrument with an external reference source, you need to set **ARBRefFreq = EXTERNAL** and specify the reference frequency provided by the external reference source here.

Important Links To learn more about the instruments supported by the ***SignalDownloader_E4438C***, please visit the E4438C and MXG websites of Agilent Technologies.

Theory of Operation

SignalDownloader_E4438C is a sub-circuit part that is built upon the ***Sink* (algorithm)** part. It utilizes the ***Sink* (algorithm)** part's capability of executing a ***MathLang Equation* (users)** when all simulation data have been collected. Here is how the operation flow goes. (To follow the following description, you can drop a ***SignalDownloader_E4438C*** part into the design/schematic and do a mouse right click on the part to bring out the drop-down menu
and select "**Open -> Model/Subcircuit**". You should see the sub-circuit design/schematic that has only a ***Sink* (algorithm)** in it. (An node named **SignalDownloader_ESG4438C(Model)** will also appear on the Workspace tree). Then do a mouse right click on this **Sink** and select **"Edit Equations"** from its drop-down menu and you will see the equation's MathLang script for this **Sink**).

First,the ***Sink* (algorithm)** in the sub-circuit is set up to write the simulation data into a file using the ***Sink* (algorithm)'s Binary** format. Note that we specifically ask the ***Sink* (algorithm)** **not to write the RF frequency into this binary file** by setting **SkipFrequency** parameter to **YES**. Once all the simulation data have been collected into the file, the **Equation** for the ***Sink* (algorithm)** will be executed. Note how the sub-circuit part passes its own **"DownloadSize"** parameter into the **Sink**'s **"BlockSize"** parameter.

The first thing this **Equation** does is to set up communication with the instrument. (Right now it can only use **LAN** to communicate). Then it will set up trivial things such as RF frequency, power, and sampling rate for the instrument. Please **note how this Equation utilize information embedded in the M_State of the Sink**. For example, the RF frequency and sampling rate information is taken from the **M_State**.

Then the **Equation** will read out the data from the binary file at the same package size specified by the **DownloadSIze** parameter and convert them into the format the instrument expects before downloading the converted data package into the instrument.

Finally, the event markers are set up if needed followed by additional SCPI commands passed in by the **SCPICommands** parameter, and then the modulation and RF output are turned ON or OFF based on the corresponding parameters' setting. Now you see why the **SCPICommands** content sets up the instrument right before modulation and RF power are turned ON/OFF.

To access the functions used in the **Equation**, make sure you have **Library Selector** pannel open via menu **View -> Advanced Windows -> Library Selector**. (You might have to close the **Part Select A** pannel to reveal the **Library Selector** pannel). Then select **Equation** for **"Library Type:"**, and **MathLang Instrument Control Functions** for **"Current Library:"**. Now you will see the list that include all the functions used by this ***SignalDownloader_E4438C*** part

SignalDownloader_N5106A Part

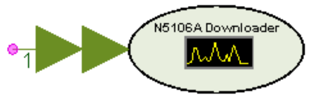
Signal downloader to download waveform and set up Agilent N5106A instrument.

Categories: *Instrumentation* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model
SignalDownloader_N5106A (algorithm)

SignalDownloader_N5106A



Description: Signal downloader to download waveform and set up Agilent N5106A instrument.
Associated Parts: *SignalDownloader N5106A Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
PXBStarted	YES: N5106A PXB is running . NO: PXB is not running, communication to PXB is supressed.: NO, YES	NO		Enumeration	NO
ConfigFile	N5106A configuration file			Filename	NO
TimeStart	Time to start waveform recording	Start_Time	s	Float	NO
TimeStop	Time to stop waveform recording	Stop_Time	s	Float	NO
ArbOn	Turn ON/OFF ARB(s) Outputs: OFF, ON	OFF	none	Enumeration	NO
NumberOfChannels	Number of ARB Channels to use: 1 IQ Channel, 2 IQ Channels, 4 IQ Channels	1 IQ Channel		Enumeration	NO
Chan1FileName	File name for downloaded waveform for IQ channel 1	chan1.bin		Text	NO
Chan1EventMarkers	Enable channel 1 event marker pulse outputs: no output; Event1 output only; Event3 output only; Event4 output only; or all outputs (Event2 is reserved for instrument internal use only): NONE, EVENT1, EVENT3, EVENT4, ALL	NONE		Enumeration	NO
Chan1MarkerStart	Sample index to start event marker pulse (Note that sample index starts at 1)	1		Integer	NO
Chan1MarkerLength	Duration of event marker pulse measured in number of waveform samples	10	none	Integer	NO
Chan2FileName	Waveform file name for IQ channel 2			Text	NO
Chan2EventMarkers	Event markers for IQ channel 2: NONE, EVENT1, EVENT3, EVENT4, ALL	NONE		Enumeration	NO
Chan2MarkerStart	Sample index to start event marker pulse for IQ channel 2	1		Positive integer	NO
Chan2MarkerLength	Marker duration measured in number of waveform samples for IQ channel 2	10		Positive integer	NO
Chan3FileName	Waveform file name for IQ channel 3			Text	NO
Chan3EventMarkers	Event markers for IQ channel 3: NONE, EVENT1, EVENT3, EVENT4, ALL	NONE		Enumeration	NO
Chan3MarkerStart	Sample index to start event marker pulse for IQ channel 3	1		Positive integer	NO
Chan3MarkerLength	Marker duration measured in number of waveform samples for IQ channel 3	10		Positive integer	NO
Chan4FileName	Waveform file name for IQ channel 4			Text	NO
Chan4EventMarkers	Event markers for IQ channel 4: NONE, EVENT1, EVENT3, EVENT4, ALL	NONE		Enumeration	NO
Chan4MarkerStart	Sample index to start event marker pulse for IQ channel 4	1		Positive integer	NO
Chan4MarkerLength	Marker duration measured in number of waveform samples for IQ channel 4	10		Positive integer	NO
ShowAdvancedParams	To display parameters used for advanced setups of the instrument: NO, YES	NO		Enumeration	NO
SCPICommands	Additonal SCPI commands before turning ARB output ON or OFF			Text	NO
DownloadSize	Waveform download size. The number of samples to be streamed into each PXB waveform file per waveform download operation	Num_Samples	none	Integer	NO
InstructionTimeout	Instrument instruction timeout limit	300	s	Float	NO
ARBRefSrc	Source for ARB reference frequency, either provided internally by the instrument (INTERNAL) or provided by an external instrument (EXTERNAL): INTERNAL, EXTERNAL	INTERNAL	none	Enumeration	NO
ARBRefFreq	ARB reference frequency (Active only when chosen EXTERNAL ARB reference)	10	MHz	Float	NO
AutoScale	Normalize waveform into [-1, 1] Volt range before downloading. YES: auto scale; NO: no auto scale.: NO, YES	YES		Enumeration	NO
ClippingLevel	The DAC clipping level for the ARB	1	V	Float	NO

Input Ports

Port	Name	Signal Type	Optional
1	ComplexIn	multiple complex	NO

Example: Bento N5106A SignalDownloader.wsv under *Instruments Examples* (examples)

Notes/Equations

You **must** have SystemVue and N5106A PXA software running in the same instrument box in order to use SignalDownloader_N5106A to communicate with N5106A PXA

It is **highly recommended** to use **Timed Complex** signal (i.e. it embodies sampling time information) for **SignalDownloader_N5106A**. Otherwise, you will need to set up the sampling rate in the PXB software yourself. If the complex IQ signal does not have sampling time information, simply use a ***SetSampleRate* (algorithm)** part in front of each input into this part to set up the sampling rate for the baseband IQ signal(s)/waveform(s).

How to determine if a signal is "Timed"? Open the **Source** part (e.g. **SineGen**, **Bits**, etc)used in the design/schematic, set **"ShowAdvancedParams = YES"**; the **"SampleRateOption"** should be **"1:Timed from SampleRate" (recommended default setting)** or **"2:Timed from Schematic"** for timed signals.

For the description of the variables **Start_Time**, **Stop_Time** and **Num_Samples**, see ***SignalDownloader_E4438C* (algorithm)**

1. **SignalDownloader_N5106A** provides an interface for streaming simulation generated baseband IQ waveforms into Bento N5106A PXB environment to produce the desired IQ baseband signals. Hence you **must** have SystemVue and N5106A PXA software running in the same instrument box in order to use it in your applications.
2. **SignalDownloader_N5106A only** accepts **Timed Complex** signal (i.e. it embodies sampling time information). If the complex IQ signal does not have sampling time information, make sure to use a ***SetSampleRate* (algorithm)** part in front of each input into this part to set up the sampling rate for the baseband IQ signal(s)/waveform(s).
3. Set **PXBStarted = YES** if you already have N5106A PXB software running since doing so will enable communication with the N5106A PXB application.
4. Use **ConfigFile** to bring in the previously saved PXB configuration file. If you don't have a N5106A configuration file, this part will configure the number of IQ channels needed based on the actual IQ inputs. Of course you can modify the configuration in PXB and save it for later use.

   Note **once the mouse cursor is in this field, on the bottom left of the parameter setup window**, a **Browse...** button is available and click it to open the file browser
5. The **NumberOfChannels** parameter controls how many additional file names and marker setup fields should be brought up since these fields are per IQ channel specific. Obviously you select the actual number of channels you are using.
6. In addition to hold the file name for the waveform for IQ channel 1, the **Chan1FileName** may also provide the base file name for the other IQ channels if you use more than one channel (i.e. **NumberOfChannels** is more than 1) while not specifying file names for the other channels. In such a case, the actual waveform file name for the other channels will be composed by using this file name with subscript such as "_1", "_2", etc.
7. The **Chan1EventMarkers** parameter controls whether a pulse will be sent out at the IQ channel 1's Event1, Event3 or Event 4 output. (Event2 is for instrument internal use only). The pulses coming out of the two outputs have the same characteristics. The pulse rises at the sample indexed by the **Chan1MarkerStart** parameter and last the number of samples determined by the **Chan1MarkerLength** parameter.
   For example, assume the waveform contains 600 samples (i.e. 600 IQ pairs), if you want a pulse to start at the 1st sample and last 10 sampling period, you simply set the **Chan1MarkerStart = 1** and **Chan1MarkerLength = 10**.
   The markers for additional IQ channels are set up the same way. **Markers are independent among IQ channel**.
8. The **SCPICommands** parameter holds additional instrument control SCPI commands you want to send to the instrument. The SCPI commands entered here will be executed right before the IQ channels are turned on.
9. The **DownloadSize** parameter determines the size of data package the simulator will write to the waveform file per data writing/streaming operation. If the size is large, a sizable amount of memory will be required to hold the package before writing it to the waveform. If the data package is small, there will be many writing operations, which might slow down the process.
   The default size **Num_Samples**, which is the total number of samples created by the simulation, works well if it is less than 1M samples. If the total number of samples is larger than 1M samples, 1M samples seems a good compromise depending on the overall memory available. Note that if multiple IQ channels are used, the memory consumption will be the multiplication of the **DownloadSize** and the **NumberOfChannels**.
10. The **AutoScale** parameter will force a normalization of the waveform generate to within [-1, 1]V range. The scaling is linear and the maximum magnitude of the I and Q data is scaled to -1V or 1V. If you choose not to use auto-scale, you can notify the simulator about the clipping level of the DAC used in the instrument by entering it in the **ClippingLevel** field, so that if the simulation waveform sample exceeds this clipping level, a warning message will be issued.
11. If an instruction can not be completed within the time specified by the **InstructionTimeout** parameter, either due to IO communication issues or due to hardware failures, a timeout will occur so that the simulator will not appear to hang.
12. When you choose to hook your instrument with an external reference source, you need to set **ARBRefFreq = EXTERNAL** and specify the reference frequency provided by the external reference source here.

Important Links

1. Learn more about the PXB/N5106A instrument from Agilent Technologies.
2. To download the N5106A PXB MIMO Receiver Tester SW to run on your PC in simulated mode, visit the download site from Agilent Technologies.

Theory of Operation

SignalDownloader_N5106A is a sub-circuit part that is built upon the ***Sink* (algorithm)** part. It utilizes the ***Sink* (algorithm)** part's capability of executing a ***MathLang Equation* (users)** when all simulation data have been collected. Additionally, it uses ***Sink* (algorithm)'s** capabilty to write data from each inputs into sepereate file. Here is how the operation flow goes. (To follow the following description, you can drop a ***SignalDownloader_N5106A*** part into the design/schematic and do a mouse right click on the part to bring out the drop-down menu
and select "**Open -> Model/Subcircuit**". You should see the sub-circuit design/schematic that has only a ***Sink* (algorithm)** in it. (An node named **SignalDownloader_N5106A(Model)** will also appear on the Workspace tree). Then do a mouse right click on this **Sink** and select **"Edit Equations"** from its drop-down menu and you will see the equation's MathLang script for this **Sink**).

First,the ***Sink* (algorithm)** in the sub-circuit is set up to write the simulation data from each complex IQ input into into a seperate waveform file using the ***Sink* (algorithm)'s** **N5106A** format. Once all the simulation data have been collected into the file, the **Equation** for the ***Sink* (algorithm)** will be executed. Note how the sub-circuit part passes its own **"DownloadSize"** parameter into the **Sink**'s **"BlockSize"** parameter.

The first thing this **Equation** does is to set up communication with the instrument by communicating to **localhost**. (This is why for applications using ***SignalDownloader_N5106A***, **SystemVue** **must** run on the same machine where the **N5106A PXB** runs). Then it will sort out the waveform file names assigned to each IQ channel and rename the waveform file names used by the **Sink** if necessary. It'll also set up trivial things such as sampling rate for the instrument. Please **note how this Equation utilizes information embedded in the M_State of the Sink**. For example, the sampling rate information is taken from the **M_State**.

Finally, the event markers are set up if needed followed by additional SCPI commands passed in by the **SCPICommands** parameter, and then the outputs of active IQ channels are turned ON or OFF based on the corresponding parameters' setting. Now you see why the **SCPICommands** content sets up the instrument right before outputs are turned ON/OFF.

To access the functions used in the **Equation**, make sure you have **Library Selector** pannel open via menu **View -> Advanced Windows -> Library Selector**. (You might have to close the **Part Select A** pannel to reveal the **Library Selector** pannel). Then select **Equation** for **"Library Type:"**, and **MathLang Instrument Control Functions** for **"Current Library:"**. Now you will see the list that include all the functions used by this ***SignalDownloader_N5106A*** part

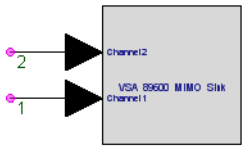
VSA_89600_MIMO_Sink Part

Categories: *Instrumentation* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
VSA_89600_MIMO_Sink (algorithm)	Agilent 89600 Vector Signal Analyzer

VSA_89600_MIMO_Sink



Description: Agilent 89600 Vector Signal Analyzer
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *VSA 89600 MIMO Sink Part* (algorithm)

Model Parameters

Name	**Description**	**Default**	**Units**	**Type**	**Runtime Tunable**
VSATitle	Text for VSA title bar	MIMO Simulation output		Text	NO
NumberOfChannels	Number of channels.: 2 channels, 3 channels, 4 channels	2 channels		Enumeration	NO
SamplesPerSymbol	Digital demodulation samples per symbol; NOT to be confused with VSA points per symbol	0.0		Float	NO
Start	Sample number to start measuring	0		Integer	NO
Stop	Sample number to stop measuring	Num_Samples - 1		Integer	NO
SetupFile	Name of measurement setup file to recall			Filename	NO
SetFreqProp	Set VSA 89600 measurement properties such as center frequency, span/sampling rate, zoom, ect. based on user inputs and simulation data.: NO, YES	YES		Enumeration	NO
RestoreHW	YES to restore VSA hardware selection at end of simulation; NO to not: NO, YES	NO		Enumeration	NO
ContinuousMode	YES enables continuous simulation; NO disables: NO, YES	NO		Enumeration	NO
RecordMode	YES enables VSA 89600 Recording mode; NO disables; Stop must be > Start, ContinuousMode = NO: NO, YES	NO		Enumeration	NO

Important Note: Starting from SystemVue2009.08 release, in order to speed up the VSA start-up, **all VSA's** created by SystemVue will remain running in the background even when their windows are closed by the user. They will be closed together on exiting SystemVue. You can manually close them all via menu *Action -> Exit Vector Signal Analyzer (89600 VSA) * or close them selectively through the window's task manager.

VSA 89600 MIMO Sink is for **MIMO applications only**. It is designed to stream the **time domain data** from multiple simulated RF antennas into VSA SW to be processed by it.

To stream single antenna data into VSA SW, please use ***VSA_89600_Sink* (algorithm)**
.

VSA MIMO Sink UI Properties

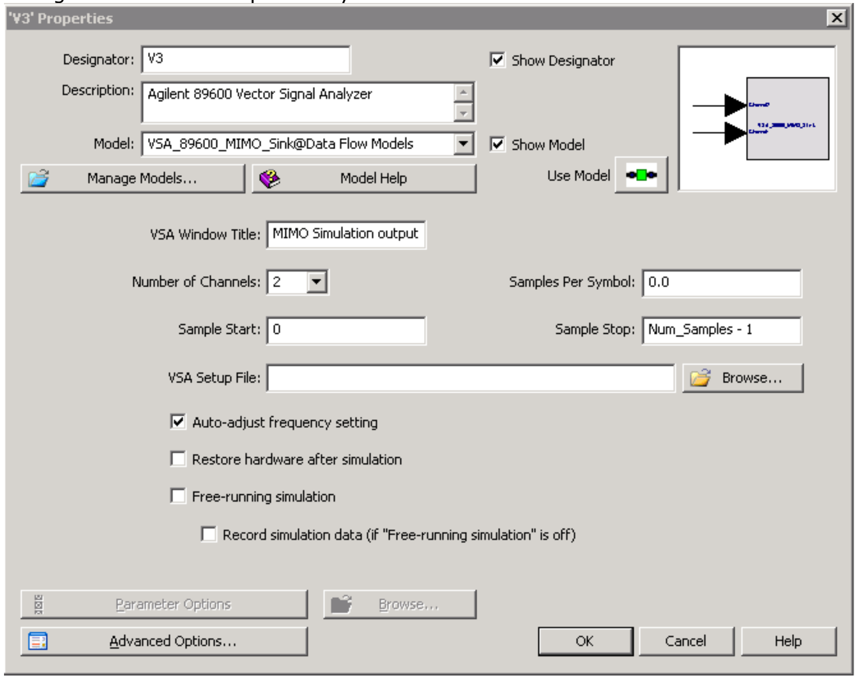
If desired, click the **Advanced Options ...** button on the bottom left corner of ***VSA MIMO Sink Properties UI*** to view the parameter list. The mapping between the parameters in the parameter list and the setup controls shown in the Graphic User Interface described above should be straightforward.

Most of the setup controls in the User Interface is self explanatory. In the following we only cover a few key setup controls.

1. **Number of Channels** Currently only up to 4 MIMO channels are supported.
2. **Sample start** & **Sample Stop** Defines the range of simulation data that VSA MIMO Sink should capture and process. To have it capture and process simulation data non-stop, check the **Free-running simulation** box.
3. **VSA Setup File:** Browse the VSA setup file that configures how to process the data

captured.

4. **Auto-adjust frequency setting** This check box maps to the parameter **SetFreqProp**. When checked, VSA 89600 measurement properties such as center frequency, span/sampling rate, zoom, etc. will be set based on simulation data.
5. **Restore hardware after simulation** By default (unchecked), the VSA application remains in its "Stream hardware" setup at the end of a simulation. It can be returned to using other VSA hardware setups through the **Hardware -> Utilities** menu on the VSA application. Once checked, the hardware setup is saved at the beginning of the simulation and then restored at the end of the simulation.
   Since some measurement setups change when the hardware setup changes, this setting can interfere with the continuity between simulations. In general, leave it unchecked (default) or specifying a VSA Setup File can have the least problems.
   One need for restoring hardware is when changing between simulations where a VSA with the same Name is used as a source in one simulation and a sink in another and the VSA instance is not closed between the simulations. In this case, the sink VSA restores the source hardware setting at the end of the simulation. Otherwise, the VSA is left configured with Stream hardware and does not function correctly as a source. The same problem arises if you would like to alternate between using the same running VSA instance as a sink in a simulation and then would like to make hardware based measurements.
6. **Free-running simulation** Once checked, the simulation will continuously run until user clicks the **Stop** button in the simulation progress display window to stop the simulation.
7. **Record simulation data** (This choice is available only when **Free-running simulation** is unchecked. The 89600 Record buffer may need additional settling points under some conditions, such as when the span is reduced (by lowering the sampling rate for example). (Note that the recording file is specified via the VSA 89600 SW's menu **"Input -> Recording ..."**).
8. Multiple VSA 89600 components (such as *VSA_89600_Source* (algorithm), *VSA_89600_MIMO_Source* (algorithm), *VSA_89600_Sink* (algorithm) and *VSA_89600_MIMO_Sink* (algorithm) can be active in a simulation and you can configure each one independently


'V3' Properties

Designator: V3
[✓] Show Designator
Description: Agilent 89600 Vector Signal Analyzer
Model: VSA_89600_MIMO_Sink@Data Flow Models
[✓] Show Model
Use Model

VSA Window Title: MIMO Simulation output
Number of Channels: 2
Samples Per Symbol: 0.0
Sample Start: 0
Sample Stop: Num_Samples - 1
VSA Setup File:

[✓] Auto-adjust frequency setting
[ ] Restore hardware after simulation
[ ] Free-running simulation
[ ] Record simulation data (if "Free-running simulation" is off)

Important Links

1. Download latest VSA SW with **14 day trial license** from Agilent Technologies
2. Purchase VSA license from Agilent Technologies
3. If you did NOT install the Agilent IO library during the VSA SW installation, you can also download Agilent IO Library yourself.
4. Also see related parts:
   VSA_89600_Sink (algorithm)
   VSA_89600_Source (algorithm)
   VSA_89600_MIMO_Source (algorithm)

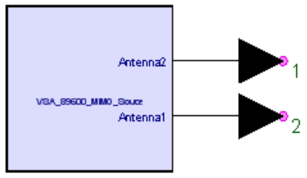
VSA_89600_MIMO_Source Part

Categories: *Instrumentation* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
VSA_89600_MIMO_Source (algorithm)	Agilent 89600 Vector Signal Analyzer for MIMO Applications

VSA_89600_MIMO_Source



Description: Agilent 89600 Vector Signal Analyzer for MIMO Applications
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *VSA 89600 MIMO Source Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
VSATitle	Text for VSA title bar	Simulation source		Text	NO	
RepeatData	VSA hardware measurement control: Repeat, Reacquire, Single pass	Reacquire		Enumeration	NO	
Pause	Pause for user to manually adjust VSA settings: NO, YES	NO		Enumeration	NO	
SetupFile	Name of setup file to recall into VSA			Filename	NO	
SetupUse	VSA setup file recall control: Always, Once, No	Always		Enumeration	NO	
SampleRateOption	Set the sample rate for the output signals: From Traces, User Specified	From Traces		Enumeration	NO	
SampleRate	Sample rate sepcified by user	Sample_Rate	Hz	Float	NO	
NumberOfAntennas	Number of receiver attennas: 2 Antennas, 3 Antennas, 4 Antennas	2 Antennas		Enumeration	NO	
Antenna1_Trace	VSA trace that will supply the 1st antenna's time domain trace.: A, B, C, D, E, F	A		Enumeration	NO	
Antenna2_Trace	VSA trace that will supply the 2nd antenna's time domain trace.: A, B, C, D, E, F	B		Enumeration	NO	
Antenna3_Trace	VSA trace that will supply the 3rd antenna's time domain trace.: A, B, C, D, E, F	C		Enumeration	NO	
Antenna4_Trace	VSA trace that will supply the 3rd antenna's time domain trace.: A, B, C, D, E, F	D		Enumeration	NO	
RecordingFile	Name of recording file to recall into VSA to be played back			Filename	NO	
AutoCapture	Capture VSA input data at start-up: NO, YES	NO		Enumeration	NO	
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO	
DefaultHardware	Use VSA default hardware configuration: NO, YES	NO		Enumeration	NO	
SetFrequencySpan	Enables the FrequencySpan parameter.: NO, YES	NO		Enumeration	NO	
FrequencySpan	If non-zero, set the frequency span.	0	Hz	Float	NO	[0:∞)
SetCenterFrequency	Enables the CenterFrequency parameter.: NO, YES	NO		Enumeration	NO	
CenterFrequency	Center frequency	0	Hz	Float	NO	(-∞:∞)
SetRange	Enables the Range parameter.: NO, YES	NO		Enumeration	NO	
Range	If non-zero and RecordingFile not set, set the range.	0	V	Float	NO	[0:∞)
RecordingLength	If non-zero and RecordingFile not set, this parameter set the recording length.	0	s	Float	NO	[0:∞)
GapOut	Enable outputting measurement Gap flag.: NO, YES	NO		Enumeration	NO	

⚠ **Important Note:** Starting from SystemVue2009.08 release, in order to speed up the VSA start-up, **all VSA's** created by SystemVue will remain running in the background even when their windows are closed by the user. They will be closed together on exiting SystemVue. You can manually close them all via menu *Action -> Exit Vector Signal Analyzer (89600 VSA) * or close them selectively through the window's task manager.

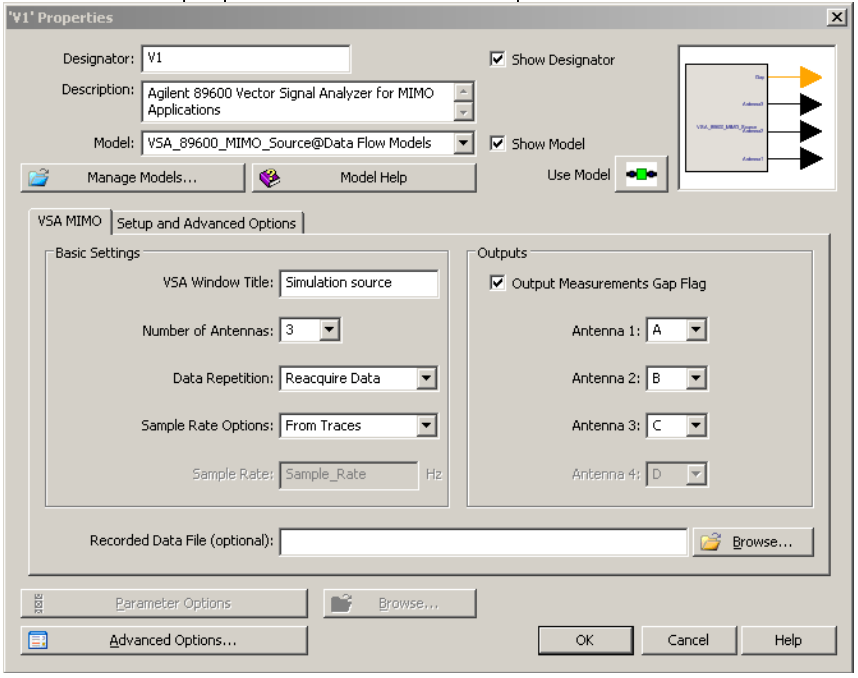
⚠ VSA 89600 MIMO Source is for **MIMO applications only**. It is designed to retrieve the **time domain data** of multiple RF sources captured by VSA SW and stream them into simulation for measurements such as Bit Error Rate measurement.

⚠ To retrieve other measurement data (e.g. spectrum) from VSA SW, please use ***VSA_89600_Source* (algorithm)**.

VSA MIMO Main Properties

1. **VSA Window Title** If the VSA SW's window is displayed (by check the **Pause VSA** box under the **VSA MIMO Advanced Properties** tab), this will be the title for that window.
2. **Number of Antennas** Controls how many RF signals are concurrently captured and streamed into the simulator from the VSA SW. Note how this parameter changes the content grouped under the "Outputs" and how it also changes the number of output pins on the little icon in the top right corner of this window.

3. **Data Repetition** It controls how data are acquired by VSA 89600 SW. Since the number of data called for by the simulator and the number of data acquired per measurement made by the VSA SW may be different, this parameter controls how data acquired by VSA SW should be used and whether VSA SW should make additional measurements to feed the data needs of the simulator.
   Single Pass: data are acquired from a single measurement by VSA per simulation.
   Repeat Data:* data are acquired from a single measurement and repetitively used by the simulation if the simulation asks for more data than what have been acquired by VSA.
   Reacquire Data: data are repeatedly acquired and fed to the simulator if the simulation requires more data than that can be captured by one measurement. Note that in the **Repeat Data** and **Reacquire Data** modes time-based data may not be continuous across measurements.
4. **Sample Rate Option**: The data acquired by the VSA SW has its own sampling rate that is determined by the measurement setup.
   From Traces: The sampling rate used for the simulation is the VSA's sampling rate.
   User Specified: Allows a sampling rate to be entered into the **Sample Rate** field.
5. **Sample Rate**: Used to specify a sampling rate that is different from what VSA measurement uses.
6. **Recorded Data File (Optional)**: If data are read in from some recorded data file (e.g. SDF data file) in stead of be acquired through "live" measurements via VSA SW, use this file browser to locate the data file.
7. **Output Measurements Gap Flag**: When simulation requires more data than that can be captured by one VSA measurement, multiple measurements need to be made and the aggregated ("stitched") data are not continuous, i.e. there will be time gaps in the data stream. This check box allows an additional "Gap" flag out put to notify whether a gap happened in between 2 captured data. If the value is 0, no gap happened. If it is 1, gap occurred. To overcome gap, use the **Automatic Capture** check box described later.
8. **Antenna1/2/3/4**: This is to specify data from which trace from VSA SW are used to feed which output pin of the VSA 89600 MIMO part.


'V1' Properties

Designator: V1 [✓ Show Designator]

Description: Agilent 89600 Vector Signal Analyzer for MIMO Applications

Model: VSA_89600_MIMO_Source@Data Flow Models [✓ Show Model]

Basic Settings

VSA Window Title: Simulation source

Number of Antennas: 3

Data Repetition: Reacquire Data

Sample Rate Options: From Traces

Sample Rate: Sample_Rate Hz

Outputs

[✓ Output Measurements Gap Flag]

Antenna 1: A

Antenna 2: B

Antenna 3: C

Antenna 4: D

Recorded Data File (optional):

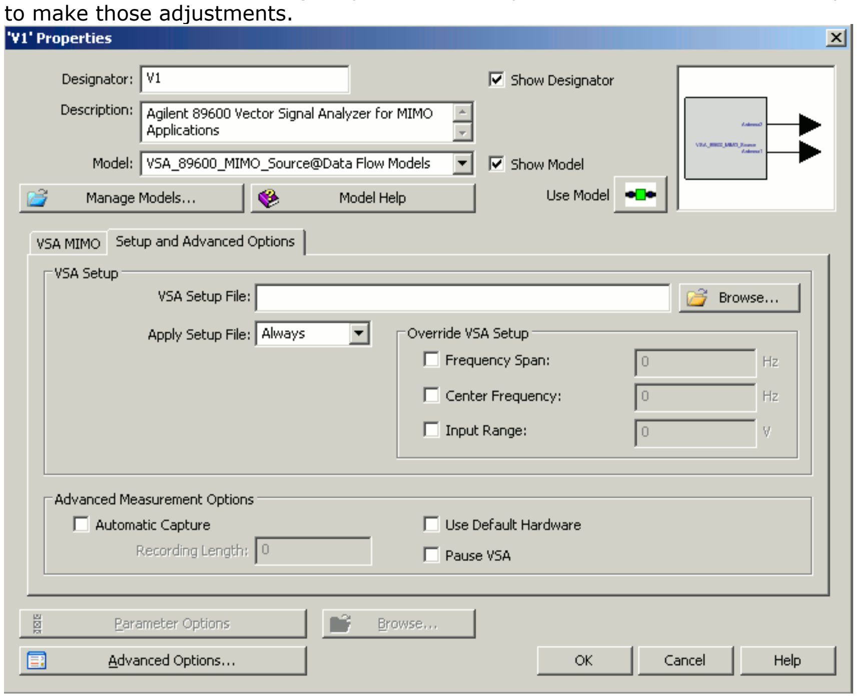
VSA MIMO Advanced Properties

1. **VSA Setup File**: A VSA setup file specified here will be recalled automatically during simulation start-up. The VSA measurement setup file is saved from the VSA application file menu, File -> Save -> Save Setup. To adjust VSA setup after a setup file has been loaded, check the **Pause VSA** box.
2. **Apply Setup File**:
   Always Setup file will be reloaded per simulation
   Once Setup file will be loaded on the first simulation.
   No Never load the setup file.

   Under current SystemVue releases, since the VSA SW will be restarted per simulation, Options **Always** and **Once** are equivalent. But this may change in the future releases.
3. **Override VSA Setup**: The VSA setups specified here will be applied to adjust VSA setup after the setup file has been loaded.
   Input Range is to adjust the maximum expected voltage range of the signal being measured.
4. **Automatic Capture** If the data required by the simulation is more than the data

held by each VSA measurement (i.e. its corresponding trace(s)), multiple measurements are required to provide the required number of data and in general, it will create measurement gaps that will distort the final measurement results such as Bit Error Rate (BER). If gap such introduced is not tolerable by the application such as BER, the solution is to have the VSA SW **record** enough data for the simulation through one **single continuous** measurement then feed these recorded data to the simulation. This check box is used to achieve this. You can leave the **Recording Length** as 0 if the setup file already has a long recording length specified.

5. **Use Default Hardware** If checked, it will have the 89600 automatically select an appropriate hardware configuration. Otherwise, the 89600 uses previous hardware selections. Note that when a source 89600 works together with a ***VSA_89600_Sink* (algorithm)** part, it can be left set up with Stream input hardware. This problem can be solved via the RestoreHW parameter of the ***VSA_89600_Sink* (algorithm)** component..
6. **Pause VSA**: If you need to make some modifications to the VSA setup after loading a VSA setup file bu tbefore simulation starts, you can check this check box so that the VSA window will be brought up after the setup file has been loaded to allow you to make those adjustments.


'V1' Properties

Designator: V1
[✓] Show Designator
Description: Agilent 89600 Vector Signal Analyzer for MIMO Applications
Model: VSA_89600_MIMO_Source@Data Flow Models
[✓] Show Model
Use Model

VSA Setup
VSA Setup File:
Apply Setup File: Always
Override VSA Setup
[ ] Frequency Span: 0 Hz
[ ] Center Frequency: 0 Hz
[ ] Input Range: 0 V

Advanced Measurement Options
[ ] Automatic Capture
Recording Length: 0
[ ] Use Default Hardware
[ ] Pause VSA

If desired, click the **Advanced Options ...** button on the bottom left corner of **VSA MIMO Main Properties** to view the parameter list. The mapping between the parameters in the parameter list and the setup controls shown in the 2 Graphic User Interfaces described above should be straightforward.

Important Links

1. Download latest VSA SW with **14 day trial license** from Agilent Technologies
2. Purchase VSA license from Agilent Technologies
3. If you did NOT install the Agilent IO library during the VSA SW installation, you can also download Agilent IO Library yourself.
4. Also see related parts:
   VSA_89600_Sink (algorithm)
   VSA_89600_Source (algorithm)

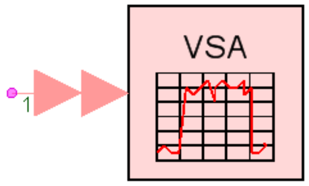
VSA_89600_Sink Part

Categories: *Instrumentation* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
VSA_89600_Sink (algorithm)	Agilent 89600 Vector Signal Analyzer

VSA_89600_Sink



Description: Agilent 89600 Vector Signal Analyzer
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *VSA 89600 Sink Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
VSATitle	Text for VSA title bar	Simulation output		Text	NO
SamplesPerSymbol	Digital demodulation samples per symbol; NOT to be confused with VSA points per symbol	0.0		Float	NO
SetupFile	Name of measurement setup file to recall			Filename	NO
SetFreqProp	Set VSA 89600 measurement properties such as center frequency, span/sampling rate, zoom, ect. based on user inputs and simulation data.: NO, YES	YES		Enumeration	NO
RestoreHW	YES to restore VSA hardware selection at end of simulation; NO to not: NO, YES	NO		Enumeration	NO
Start	Sample number to start measuring	0		Integer	NO
Stop	Sample number to stop measuring	Num_Samples - 1		Integer	NO
ContinuousMode	YES enables continuous simulation; NO disables: NO, YES	NO		Enumeration	NO
RecordMode	YES enables VSA 89600 Recording mode; NO disables; Stop must be > Start, ContinuousMode = NO: NO, YES	NO		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	multiple anytype	NO

⚠ **Important Note:** Starting from SystemVue2009.08 release, in order to speed up the VSA start-up, **all VSA's** created by SystemVue will remain running in the background even when their windows are closed by the user. They will be closed together on exiting SystemVue. You can manually close them all via menu *Action -> Exit Vector Signal Analyzer (89600 VSA) * or close them selectively through the window's task manager.

⚠ The input **must** be **timed**, either **timed complex** IQ for BaseBand or **timed Complex Envelope**. If needed, consider using a ***SetSampleRate* (algorithm)** part to add the sampling rate/time information and/or using a ***Modulator* (algorithm)** part to add the RF carrier frequency.
Example: VSA89600 Demod QPSK.wsv under *Instruments Examples* (examples)

Notes/Equations

1. The VSA_89600 models provide a dynamic link to integrate the 89600 series VSA software with SystemVue.
   - Before using this model, the VSA89600 software must be installed. The VSA 89600 software can be downloaded from http://www.agilent.com/find/89600 .
   - For 89600 analyzer reference information, example measurements, or a getting started video, access Help or choose **Start -> Programs -> Agilent 89600 VSA -> Help**.
2. The VSA_89600_Sink model provides a stream interface where you can input digitized waveforms directly from SystemVue to 89600-series VSA software without using the 89600-series hardware. The full functionality of the 89600 analyzer is available to analyze and display the SystemVue signal.
3. The **SamplesPerSymbol** parameter provides a convenient way to set the analyzer's

Symbol Rate. This is different from the analyzer's Points/Symbol parameter, which adjusts the analyzer's interpolation of demodulated data. SamplesPerSymbol takes precedence over the Symbol Rate in the setup file, if a SetupFile is specified.

4. The analyzer's digital demodulation algorithm limits the analyzer's span to a maximum of 15.625 times the Symbol Rate. This, plus the analyzer's maximum decimation rate, places a lower limit on the Symbol Rate.
5. The **SetupFile** parameter can be used to recall a VSA setup file automatically during simulation start-up. The VSA measurement setup file, **SetupFile**, is saved from the VSA application file menu, **File -> Save -> Save Setup**. To refine an existing **SetupFile**, you can modify the VSA application while simulating and then save the setup file before restarting the simulation.

   Note that once the mouse cursor is in **SetupFile** parameter's field, on the bottom left of the parameter setup window, a **Browse...** button is available and clicking it will open a file browser.

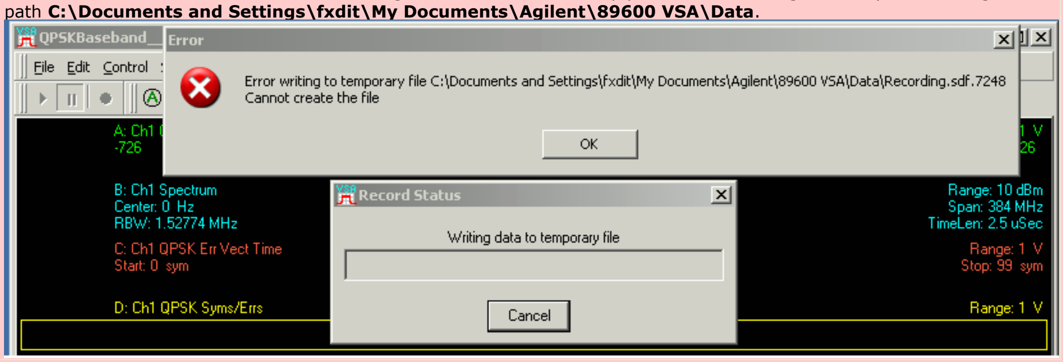
6. Triggering is not available in the 89600 when the input is from a simulation.
7. If **SetFreqProp = NO**, the VSA's measurement center frequency, span, and zoom mode are left unchanged (except that these are loaded from a setup file, if **SetupFile** is specified). If **SetFreqProp = YES**, the VSA's measurement center frequency, span, and zoom parameters is set to those of the simulation, overriding those in the setup file, if specified.
8. Unless a setup file is specified, the VSA application begins the simulation with its previous setup, with a few modifications dictated by the simulation. This setting enables for continuity of the VSA setup between simulations. **RestoreHW = YES** can interfere with this. This continuity means that the initial VSA application setup can be important when a setup file is not specified. This includes the Hardware Setup. Starting a simulation in a known state, such as a preset state, can resolve some VSA initialization problems. The VSA can be preset in various ways under **File -> Preset** on the VSA application. The Simulate Hardware selection under **Hardware -> Utilities** can be a useful starting point.
9. If **RestoreHW = NO** (default), the VSA application remains in its Stream hardware setup at the end of a simulation. It can be returned to using other VSA hardware setups through the **Hardware -> Utilities** menu on the VSA application.
10. If **RestoreHW = YES**, the hardware setup is saved at the beginning of the simulation and then restored at the end of the simulation. Since some measurement setups change when the hardware setup changes, this setting can interfere with the continuity between simulations as described in previous note. In general, setting RestoreHW to NO or specifying SetupFile can have the least problems.
    One need for **RestoreHW = YES** is when changing between simulations where a VSA with the same Instance Name is used as a source in one simulation and a sink in another and the VSA instance is not closed between the simulations. In this case, the sink VSA restores the source hardware setting at the end of the simulation. Otherwise, the VSA is left configured with Stream hardware and does not function correctly as a source. The same problem arises if you would like to alternate between using the same running VSA instance as a sink in a simulation and then would like to make hardware based measurements.
11. Set **ContinuousMode** to **"YES"** if you want the simulation to run indefinitely. You can always click the **Stop** button in the simulation progress display window to stop the simulation.
12. When **RecordMode = YES** (i.e., Time Capture mode, which is available only when **ContinuousMode = NO**), the 89600 Record buffer may need additional settling points under some conditions, such as when the span is reduced (by lowering the sampling rate for example). (Note that the recording file is specified via the VSA 89600 SW's menu **"Input -> Recording ..."**).
13. Multiple **VSA_89600_Sink** components can be active in a simulation and you can configure each one independently
14. When a simulation that contains an active VSA_89600_Sink component starts, it attaches to a running 89600 whose title begins with the component's instance name. If no such 89600 instance is found, a new 89600 is created, and its title is set to the associated component's instance name and **VSATitle** text.

Important Links

1. Download latest VSA SW with **14 day trial license** from Agilent Technologies
2. Purchase VSA license from Agilent Technologies
3. If you did NOT install the Agilent IO library during the VSA SW installation, you can also download Agilent IO Library yourself.

Commonly Seen Errors

1. Error Seen When Using Workspace Created on a Different Computer
   If you copy a workspace from a different computer, occasionally you might see the following error messages:
   "Cannot create the file C:\Documents and Settings\fxdit\My Documents\Agilent\89600 VSA\Data\Recording.sdf.4088", recording file ...
   To fix this, you can either try to replace the **VSA_89600_Sink** used in the schematic/design with a new one, or simply add all the missing folders specified along the path **C:\Documents and Settings\fxdit\My Documents\Agilent\89600 VSA\Data**.
2. Error Seen When Using **RecordMode**
   When you set **"RecordMode"** to **"YES"** and you see the following **Error writing to temporary file** **C:\Documents and Settings\fxdit\My Documents\Agilent\89600 VSA\Data\Recording.sdf.1234** complaint from VSA 89600 SW, it is very likely you have copied the workspace (project) from somewhere else. To fix it, you can either try to replace the **VSA_89600_Sink** used in the schematic/design with a new one, or simply add all the missing folders specified along the path **C:\Documents and Settings\fxdit\My Documents\Agilent\89600 VSA\Data**.



Error writing to temporary file C:\Documents and Settings\fxdit\My Documents\Agilent\89600 VSA\Data\Recording.sdf.7248
Cannot create the file

Writing data to temporary file

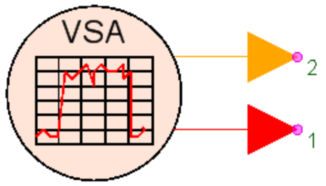
VSA_89600_Source Part

Categories: *Instrumentation* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
VSA_89600_Source (algorithm)	Agilent 89600 Vector Signal Analyzer

VSA_89600_Source



Description: Agilent 89600 Vector Signal Analyzer
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *VSA 89600 Source Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
VSATitle	Text for VSA title bar	Simulation source		Text	NO	
Pause	Pause for user to manually adjust VSA settings: NO, YES	NO		Enumeration	NO	
OutputType	Output port type: Timed, Frequency, Demod Errors, Complex Scalar, Float Scalar, Integer Scalar	Timed		Enumeration	NO	
VSATrace	VSA trace that will supply data.: A, B, C, D	B		Enumeration	NO	
RepeatData	VSA hardware measurement control: Repeat, Reacquire, Single pass	Repeat		Enumeration	NO	
TStep	Simulation time step	0	s	Float	NO	
SetupFile	Name of setup file to recall into VSA			Filename	NO	
SetupUse	VSA setup file recall control: Always, Once, No	Always		Enumeration	NO	
RecordingFile	Name of recording file to recall into VSA to be played back			Filename	NO	
AutoCapture	Capture VSA input data at start-up: NO, YES	NO		Enumeration	NO	
DefaultHardware	Use VSA default hardware configuration: NO, YES	NO		Enumeration	NO	
FrequencySpan	If non-zero, set the frequency span.	0	Hz	Float	NO	[0:∞)
SetCenterFrequency	Controls the CenterFrequency parameter.: NO, YES	NO		Enumeration	NO	
CenterFrequency	Center frequency	0	Hz	Float	NO	(-∞:∞)
Range	If non-zero and RecordingFile not set, set the range.	0	V	Float	NO	[0:∞)
RecordingLength	If non-zero and RecordingFile not set, this parameter set the recording length.	0	s	Float	NO	[0:∞)

Output Ports

Port	Name	Description	Signal Type	Optional
1	out	Measurement data	anytype	NO
2	gap	VSA input data gap signal	int	NO

⚠ **Important Note:** Starting from SystemVue2009.08 release, in order to speed up the VSA start-up, **all VSA's** created by SystemVue will remain running in the background even when their windows are closed by the user. They will be closed together on exiting SystemVue. You can manually close them all via menu *Action -> Exit Vector Signal Analyzer (89600 VSA) * or close them selectively through the window's task manager.

In SystemVue2009.05 official release, we have taken out the multiple time domain traces (for MIMO applications) support we introduced during SystemVue2009.05 Alpha. If you are using this feature, please use *VSA 89600 MIMO Source Part* (algorithm) instead.

Measurement data from VSA 89600 streams out of **Port 1**

Example: VSA89600 Recall QAM512 Data.wsv under *Instruments Examples* (examples)

Notes/Equations

1. The VSA_89600 models provide a dynamic link to integrate the 89600 series VSA software with SystemVue.
2. Before using this model, the VSA 89600 software must be installed. The VSA 89600 software can be downloaded from http://www.agilent.com/find/89600 .
3. For 89600 analyzer reference information, example measurements, or a getting started video, access **Help** or choose **Start -> Programs -> Agilent 89600 VSA -> Help**.
4. The VSA_89600_Source model transfers measurement data from the 89600 Vector Signal Analyzer. The model outputs different amounts and types of data depending on OutputType, as described in detail under the **OutputType** parameter.
5. Measurements can be acquired from any 89600 trace and are transferred in raw (unformatted) state.
6. The **VSATitle** parameter sets the title in the 89600 window title bar.
7. The **Pause** parameter controls the 89600 start-up sequence:
   - When **Pause = YES**, the simulation displays a message dialog and pause before acquiring measurement data from the 89600. This allows you to configure the 89600 before pressing the OK button to proceed.
8. The **OutputType** parameter establishes the configuration of the model's output data port.

   Important
   - The data type of the 89600 trace selected by **VSATrace** **must** be compatible with the output type.

   - **Timed** - Timed data is output at the 89600 center frequency and timestep (refer to **TStep** for detail). The data can be complex or baseband, depending on the 89600 zoom state. Requires time domain measurements from the 89600.
   - **Frequency** - Spectrum data is output as pairs of complex numbers. The first number is the real frequency (the imaginary part is zero), and the second number is the complex voltage at that frequency. You can connect a to the VSA_89600_Source data port to separate the signal. Requires frequency domain measurements from the 89600. Example 89600 trace data type: Spectrum.
   - **Demod Errors** - Demodulation error data is output as sets of floating-point (real) values. The number of values and required 89600 configuration vary with demod type. See note 20.
   - **Complex Scalar** - Complex numbers are output. If the 89600 measurement data is real-valued, the imaginary part of the output values is zero. Example 89600 trace data type: "Error Vector Time" in Digital Demod mode.
   - **Float Scalar** - Floating point numbers are output. Requires real-valued 89600 measurement data. Example 89600 trace data type: "IQ Mag Error" in Digital Demod mode.
   - **Integer Scalar** - Integer numbers are output. Requires real-valued 89600 measurement data. Useful for sourcing demodulation symbols when the 89600 trace data type is Syms/Errs.
9. The **VSATrace** parameter specifies which 89600 trace provides measurement data.
10. The **RepeatData** parameter controls the transfer of data from the 89600 during hardware-based measurement.
    - When **RepeatData = Single pass**, the model supplies data from a single measurement. The **Repeat** option acquires a single measurement and repetitively sources it into the simulation. **Reacquire** repeatedly acquires and sources new measurements. Note that in the Repeat and Reacquire modes time-based data is not continuous across measurements.

      When 89600 input is from a recording the RepeatData setting is ignored. Refer to note 19 for information on operation with recordings.
11. The **TStep** parameter can be used to specify a target timestep when sourcing timed data. The model trys to adjust the 89600's span to achieve the requested step size via resampling. If the 89600 cannot be set to the required span, a warning message is output to the status window.
12. The **SetupFile** parameter can be used to recall automatically a VSA setup file during simulation start-up. The VSA measurement setup file, **SetupFile**, is saved from the VSA application file menu, **File -> Save -> Save Setup**. To refine an existing **SetupFile**, you can modify the VSA application while simulating and then save the setup file before restarting the simulation. If **Pause = YES**, the setup file is loaded before the **Pause** dialog is displayed.

    Note that once the mouse cursor is in **SetupFile** parameter's field, on the bottom left of the parameter setup window, a **Browse...** button is available and clicking it will open a file browser.
13. The **SetupUse** parameter specifies when an 89600 setup file (specified in the **SetupFile** parameter) is recalled. Options are:

- **Always** - Recall setup file on every simulation run.
- **Once** - Recall setup file only when 89600 is started.
- **No** - Do not recall setup file.

14. The **RecordingFile** parameter can be used to recall automatically a recording into the 89600 during simulation start-up. #* If **Pause = YES**, the recording file is loaded before the **Pause** dialog is displayed.

    Use the **Browse...** button to open the file browser in the same way as handling **SetupFile**.

15. The **AutoCapture** parameter can be set to **YES** to have the 89600 automatically initiate a capture recording of hardware input data at the beginning of a simulation. When the capture is complete, the model begins sourcing the recorded data. Time-based measurements is continuous (note 19). Recording length can be controlled via an 89600 setup file.
16. The **DefaultHardware** parameter can be set to YES to have the 89600 automatically select an appropriate hardware configuration. Otherwise, the 89600 uses previous hardware settings. Note that when a source 89600 reuses the same running 89600 instance that was used by a sink 89600, it can be left set up with Stream input hardware. This problem can be solved via the **RestoreHW** parameter of the ***VSA_89600_Sink* (algorithm)** component.
17. Each 89600 measurement produces a block of data points, which are pipelined to the simulation. As a result, simulation plots typically lag 89600 traces. The number of points in a block varies with 89600 measurement type and configuration. For reference, the block size is output to the status window during startup. This value can be useful when configuring other components (for example, plot persistence and update size). Also, any change in block size during simulation is logged in the status window.
18. The units associated with transferred 89600 data are output to the status window during startup. Unless otherwise indicated, values are peak units.
19. To avoid transfer of overlapped measurement data, the 89600's maximum overlap for averaging off is set to zero when a simulation starts.
20. If the model detects a change to one of the following 89600 settings while sourcing data, the model disconnects from the 89600.
    - Input data source
    - Demodulator configuration
    - Data type of the trace supplying measurements (VSATrace)
    - Sweep mode
    - Center frequency (for timed output)
    - Timestep size (for timed output)

    The recommended practice is to use the **Pause** and/or **SetupFile** options to pre-configure the 89600.

21. When a simulation that contains an active VSA_89600_Source component starts, it attaches to a running 89600 whose title begins with the component's instance name. If no such 89600 instance is found, a new 89600 is created, and its title is set to the associated component's instance name and VSATitle text.
22. When 89600 input is from a recording, the model steps through the recording, transferring all measurements. 89600 recording playback properties can be used to control start/stop points and looping. In this mode time-based data are continuous, with the exception of wrapped values when a playback loop occurs.
23. Demodulation Error/Summary information is available by configuring the trace data.

    For details on results available, please refer to the following:
    - 89600 help index topic symbol table and look for the demodulation format of interest.
    - The demod formats available depend on 89600 licensing (For example, Digital Demod requires option AYA, and cdma2000 requires option B7.)
    - The error summary data and the numbering for each demodulation format are documented in tables in the 89600 help documentation (look in the index under demod errors).
    - Also note that some demodulation formats include several of these tables. The table values are output in a fixed sequence, which can be different from the order seen on the 89600 display.
    - The number of output error values depends on the configuration.
    - The easiest way to get an individual error value is to place a DownSample component in series with the VSA89600Source output.
    - For example, consider the list of Digital Demod error items. The DownSample Factor parameter would be set to the number of error values (21). To select EVM, the Phase parameter would be set to 20 (EVM position in the list subtracted from 21).
    - When OutputType is Integer Scalar and trace data is configured to one of the above types (Syms/Errs, for example), the module sources demodulation symbols.

24. The gap output port emits a 1 when there is a discontinuity in 89600 input data. Otherwise it emits 0.
25. Multiple VSA_89600_Source components can be active in a simulation.
26. When both VSA_89600_Source and ***VSA_89600_Sink* (algorithm)** components are present in a design, configuring the ***VSA_89600_Sink* (algorithm)** for 0% time record overlap minimizes the effect of gaps in source data.
27. The VSA_89600_Source component can be used with a PSA E444xA/89601A combination. The VSA89600 software runs on a PC connected to the E444xA, via LAN or GPIB, and provides hardware control, modulation analysis, and complete results displays. The controls and display of the E4406A are disabled while operating with the 89601A software.

- For more information, refer to the Agilent PSA website http://www.agilent.com/find/psa .
- Special options required for use of the Performance Spectrum Analyzers E444xA and the VSA89600 software are described in these product notes:
  - Product Note 5988-5015EN; available on the website http://cp.literature.agilent.com/litweb/pdf/5988-5015EN.pdf .
  - Product Note 5988-4094EN; available on the website http://cp.literature.agilent.com/litweb/pdf/5988-4094EN.pdf .

28. The VSA_89600_Source component can be used with a VSA E4406A/89601A combination. The VSA89600 software runs on a PC connected to the E4406A, via LAN or GPIB, and provides hardware control, modulation analysis, and complete results displays. The controls and display of the E4406A are disabled while operating with the 89601A software.

- For more information, refer to the Agilent VSA E4406A website http://www.agilent.com/find/vsa .
- Special options required for use of the Vector Signal Analyzer E4406A and the VSA89600 software are described in Product Note 5988-2906EN, which is available on the website http://cp.literature.agilent.com/litweb/pdf/5988-2906EN.pdf .

29. The VSA_89600_Source component can be used with an ESA/89601A combination. The VSA89600 software runs on a PC connected to the ESA, via GPIB, and provides hardware control, modulation analysis, and complete results displays. The controls and display of the ESA are disabled while operating with the 89601A software.

- For more information, refer to the Agilent ESA website http://www.agilent.com/find/esa .
- Special options required for use of the ESA and the VSA89600 software are described in Product Note 5988-4097EN, which is available on the website http://cp.literature.agilent.com/litweb/pdf/5988-4097EN.pdf .

30. The VSA_89600_Source component can be used with an Infiniium oscilloscope/89601A combination. The VSA89600 software runs on a PC connected to the Infiniium scope, via LAN or GPIB, and provides hardware control, modulation analysis, and complete results displays. The controls and display of the scope are disabled while operating with the 89601A software.

Special options required for use of the Infiniium scope and the VSA89600 software are described in this product note:
http://cp.literature.agilent.com/litweb/pdf/5988-4096EN.pdf .

31. The VSA_89600_Source component does not handle averaged measurements as a special case; each 89600 analyzer trace update is output to the simulation. The default 89600 analyzer average setup has Fast Average disabled, so the traces are updated each time new measurement results are added to the average, starting with the first measurement.
32. To force only the final, fully averaged result to be output to the simulation, select **MeasSetup -> Average** on the 89600 analyzer and check both the Fast Average and the Same as Count check boxes. This prevents the analyzer from updating the screen until the selected number of measurements have been averaged together.

Additional Important Links

1. Download latest VSA SW with **14 day trial license** from Agilent Technologies
2. Purchase VSA license from Agilent Technologies
3. If you did NOT install the Agilent IO library during the VSA SW installation, you can also download Agilent IO Library yourself.

Commonly Seen Errors
Error Seen When Using Workspace Created on a Different Computer: If you copy a workspace from a different computer, occasionally you might see the following error messages if the workspace reads data from recorded **sdf** file:

- **Warning:** VSA_89600_Source `VSA89600RecallQAM512__VSA89600Source': **VSA recall recording error 0x8004021a**, **"Cannot create the file C:\Documents and Settings\fxdit\My Documents\Agilent\89600 VSA\Data\Recording.sdf**.4088", recording file **"C:\Program Files\SystemVue(Version)\Examples\Instruments\VSA89600Source\Qam512.sdf"**
- **Warning:** VSA_89600_Source `VSA89600RecallQAM512__VSA89600Source': Disconnecting from VSA...*

To fix this, you can either try to replace the **VSA_89600_Source** used in the schematic/design with a new one, or simply add all the missing folders specified along the path **C:\Documents and Settings\fxdit\My Documents\Agilent\89600 VSA\Data**.

Math Matrix

A matrix is a two dimensional or rectangular arrangement of scalar values. The two dimensions are commonly referred as the number of rows (NumRows) and the number of columns (NumCols). If the matrix is square, only the number of rows or columns (RowsCols) is specified. If NumRows and NumCols or RowsCols are not visible as parameters, they are inferred. Matrix parts are concerned with:

1. generation of a matrix
2. conversion to or from a matrix
3. operation upon a matrix

Contents

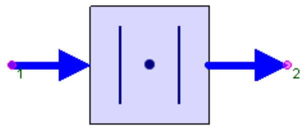
Abs_M Part

Categories: *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Abs_M (algorithm)	Absolute Value Matrix Function

Abs_M (Absolute Value Matrix Function)



Description: Absolute Value Matrix Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Abs M Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	input	real matrix	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real matrix	NO

Notes/Equations

1. The input matrix is copied to output with each element replaced with the absolute value of that element.
2. For every input matrix, an output matrix is written_

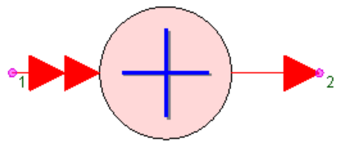
Add Part

Categories: *C++ Code Generation* (algorithm), *Math Matrix* (algorithm), *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Add (algorithm)	Multiple Input Adder
AddEnv (algorithm)	Envelope Signal Adder
AddFxp (hardware)	Fixed Point Adder

Add (Multiple Input Adder)



Description: Multiple Input Adder
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *Add Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	input	multiple anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim), *fixed point* (sim), or *variant* (sim).

Notes/Equations

1. The Add model produces the sum of the inputs at the output.
2. This model reads 1 sample from all inputs and writes 1 sample to the output.
3. For discussion on the variant type, see *Variant* (sim).
4. For a single input add, the input is copied to the output.

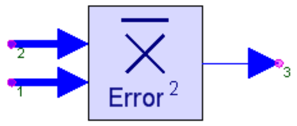
AvgSqrErr_M Part

Categories: *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AvgSqrErr_M (algorithm)	Mean Squared Error Matrix Averager

AvgSqrErr_M (Mean Squared Error Matrix Averager)



Description: Mean Squared Error Matrix Averager
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AvgSqrErr M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NumInputsToAverage	Number of input matrices to average	8		Integer	NO	[1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input1	real matrix	NO
2	input2	real matrix	NO

Output Ports

Port	Name	Signal Type	Optional
3	output	real	NO

Notes/Equations

1. The squared difference between corresponding elements of matrices from input1 and input2 are summed. These sums are calculated and then averaged over NumInputs number of inputs. The averaged sum is output.
2. For every NumInputs matrices from each input port, one non-negative float value is output.

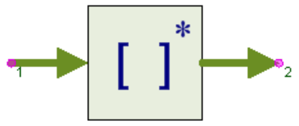
Conjugate_M Part

Categories: *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
Conjugate_M (algorithm)	Conjugate Matrix Function

Conjugate_M (Conjugate Matrix Function)



Description: Conjugate Matrix Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Conjugate M Part* (algorithm)

Input Ports

Port	**Name**	**Signal Type**	**Optional**
1	input	complex matrix	NO

Output Ports

Port	**Name**	**Signal Type**	**Optional**
2	output	complex matrix	NO

Notes/Equations

1. The input matrix is copied to output with each element replaced with the complex conjugate of that element.
2. For every input matrix, an output matrix is written.

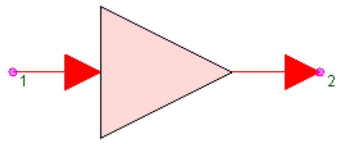
Gain Part

Categories: *C++ Code Generation* (algorithm), *Math Matrix* (algorithm), *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
Gain (algorithm)	Constant Gain
Amplifier (algorithm)	Nonlinear Amplifier with Noise Figure
GainFxp (hardware)	Fixed Point Gain
NegateFxp (hardware)	Fixed Point Negate

Gain



Description: Constant Gain
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *Gain Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Gain	Gain value	1		None	YES

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim), *fixed point* (sim), or *variant* (sim).

Notes/Equations

1. Gain multiplies the input by the Gain parameter.
2. For every input, a product is output.
3. As the Gain parameter is a variant, the product may be promoted from the input type, e.g. if the Gain parameter is a complex matrix while the input is real, the output will be a complex matrix. For discussion on the variant type, see *Variant* (sim).

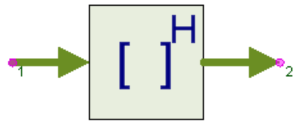
Hermitian_M Part

Categories: *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Hermitian_M (algorithm)	Hermitian Matrix Function

Hermitian_M (Hermitian Matrix Function)



Description: Hermitian Matrix Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Hermitian M Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	input	complex matrix	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	complex matrix	NO

Notes/Equations

1. The conjugate transpose of the input is output.
2. If the input is a M × N matrix, then the output is a N × M matrix.
3. Let x[i,j] be an element of the input and y[j,i] be an element of the output, then y[j,i] is set to the conjugate of x[i,j] when i is not equal to j.

See:
TransposeCx_M (algorithm)
Transpose_M (algorithm)

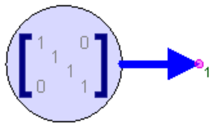
Identity_M Part

Categories: *Math Matrix* (algorithm), *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Identity_M (algorithm)	Identity Matrix Generator
IdentityCx_M (algorithm)	Complex Identity Matrix Generator

Identity_M (Identity Matrix Generator)



Description: Identity Matrix Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Identity M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
RowsCols	Number of rows and columns in output square matrix	2		Integer	NO	[2:∞)	RC
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D

Output Ports

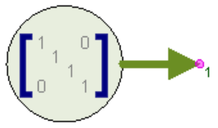
Port	Name	Signal Type	Optional
1	output	real matrix	NO

Notes/Equations

1. Identity_M is a constant square matrix source whose dimensions are specified by the RowsCols parameter.
2. The constant matrix is formed by inserting real ones into the diagonal elements beginning at element [1,1]. Zeroes are inserted at the off diagonal elements.
3. For other parameter descriptions, see *Untimed Sources* (algorithm).
4. Fill value is a real matrix zero.

See:
IdentityCx_M (algorithm)
Diagonal_M (algorithm)
DiagonalCx_M (algorithm)
Const (algorithm)

IdentityCx_M (Complex Identity Matrix Generator)



Description: Complex Identity Matrix Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Identity M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
RowsCols	Number of rows and columns in output square matrix	2		Integer	NO	[2:∞)	RC
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D

Output Ports

Port	Name	Signal Type	Optional
1	output	complex matrix	NO

Notes/Equations

1. IdentityCx_M is a constant square matrix source whose dimensions are specified by the RowsCols parameter.
2. The constant matrix is formed by inserting complex ones into the diagonal elements beginning at element [1,1]. Zeroes are inserted at the off diagonal elements.
3. For other parameter descriptions, see *Untimed Sources* (algorithm).
4. Fill value is a complex matrix zero.

See:
Identity_M (algorithm)
Diagonal_M (algorithm)
DiagonalCx_M (algorithm)
Const (algorithm)

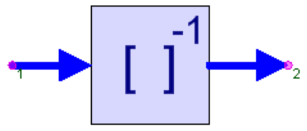
Inverse_M Part

Categories: *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Inverse_M (algorithm)	Inverse Matrix Function
InverseCx_M (algorithm)	Complex Inverse Matrix Function

Inverse_M (Inverse Matrix Function)



Description: Inverse Matrix Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Inverse M Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	input	real matrix	NO

Output Ports

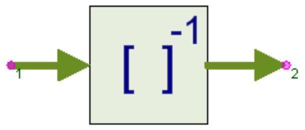
Port	Name	Signal Type	Optional
2	output	real matrix	NO

Notes/Equations

1. The output matrix is the inverse of the input matrix, i.e.
$$\left[input\right] \times \left[output\right] = \left[IdentityMatrix\right]$$
2. For every input matrix, an inverse matrix is output.
3. The input matrix must be square.

See:
InverseCx_M (algorithm)

InverseCx_M (Complex Inverse Matrix Function)



Description: Complex Inverse Matrix Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Inverse M Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	input	complex matrix	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	complex matrix	NO

Notes/Equations

1. The output matrix is the inverse of the input matrix, i.e.
$$\left[input\right] \times \left[output\right] = \left[IdentityMatrix\right]$$
2. For every input matrix, an inverse matrix is output.
3. The input matrix must be square.

See *Inverse_M* (algorithm).

Mapper_M Part

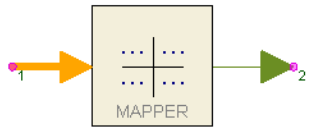
Categories: *C++ Code Generation* (algorithm), *Communications* (algorithm), *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Mapper_M (algorithm)	Most significant bit first. BPSK, QPSK, PSK8, QAM16, QAM32, QAM64, QAM128, QAM256

<#comment></#comment><#comment></#comment>

Mapper_M (Most significant bit first)



Description: Most significant bit first. BPSK, QPSK, PSK8, QAM16, QAM32, QAM64, QAM128, QAM256
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Mapper M Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	In	boolean matrix	NO

Output Ports

Port	Name	Signal Type	Optional
2	Out	complex	NO

Notes/Equations

1. The Mapper_M model maps an input boolean matrix into a complex constellation point. The modulation type is dynamically determined by the number of elements in the input matrix. The bits are grouped by reading the matrix in column-major order, with the first matrix element being interpreted as the most significant bit.
2. A constellation point is a pair of real values (I,Q) that is expressed on the output as I + jQ. Later in the modulation chain, I modulates the inphase part of the carrier, and Q modulates the quadrature part of the carrier over a symbol period.
3. Each modulation type has its constellation and symbol length. The symbol length, i.e. the number of input bits per symbol, is detailed in the following table.

Modulation Type and Symbol Length

Symbol Length (Number of matrix elements)	Modulation Type
1	*BPSK*
2	*QPSK*
3	*PSK8*
4	*QAM16*
5	*QAM32*
6	*QAM64*
7	*QAM128*
8	*QAM256*

1. For *QPSK* and *PSK8* the mapping from bits to symbols is using Gray encoding. For *QAM16*, *QAM32*, *QAM64*, *QAM128*, and *QAM256*, Gray encoding is used inside each quadrant.
2. For *BPSK*, bit value 0 is mapped to 1 + j0 and bit value 1 is mapped to -1 + j0.
3. For *QPSK*, the constellation map is illustrated in *QPSK Constellation* (algorithm). For *PSK8*, the constellation map is illustrated in *8PSK Constellation* (algorithm).
4. The symbol mappings for *QAM16*, *QAM32*, *QAM64*, *QAM128* and *QAM256* are described in the section 9 of [1], and their constellation maps are illustrated in figure 7-8 of [1].
5. QAM constellations need definition only for quadrant 1. The constellation points in quadrants 2, 3 and 4 are derived from quadrant 1 by selecting the quadrant 1 constellation value with the least significant bits of the input symbol and rotating that constellation value by the amount selected by the two most significant bits of the input symbol, $b_i b_q$, as specified in table *Conversion of Constellation Points* (algorithm).

Conversion of Constellation Points

Quadrant	Symbol Most Significant Bits ($b_i b_q$)	Rotation
1	00	0
2	10	π/2
3	11	π
4	01	3π/2

16QAM, 32QAM, 64QAM, 128QAM and 256QAM constellation maps are illustrated in *16 and 32QAM Constellation* (algorithm) through *256QAM Constellation* (algorithm).

See
Mapper (algorithm)
Demapper (algorithm)

References

1. EN 300 429, "Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for cable systems," V1.2.1, 1998-04.

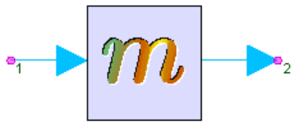
MathLang Part

Categories: *Math Matrix* (algorithm), *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
MathLang (algorithm)	Math Language Block

MathLang (Math Language Model)



Description: Math Language Block
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *MathLang Part* (algorithm)

Model Parameters

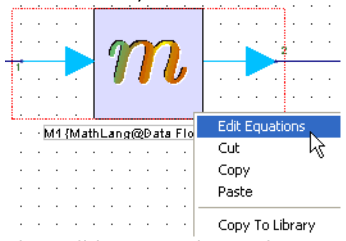
Name	Description	Default	Units	Type	Runtime Tunable

Summary

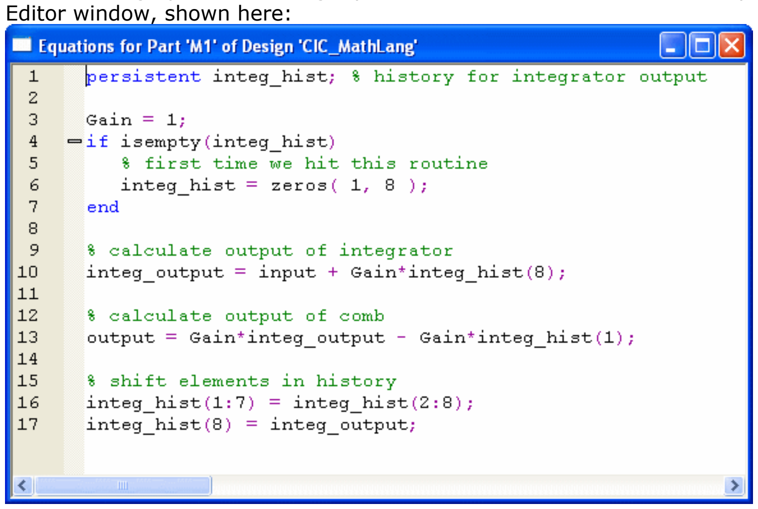
The MathLang model uses Math Language equations to process input data and produce output data. This block is designed to be capable of being a fully self-contained model. Although by default, when placed from the associated part in the Algorithm Design library, this block has a single input and a single output, this model is compatible with any symbol. Data presented to input terminals can be obtained and used in MathLang equations which then assigns data to the output terminals of the symbol being used. Custom Parameters and/or State Variables can be defined and are fully visible from the MathLang equations.

MathLang Equations

There are two ways to view and edit the MathLang equations associated with a part using the MathLang model. The preferred way is to right-click the part in the schematic and select *Edit Equations* as shown here:



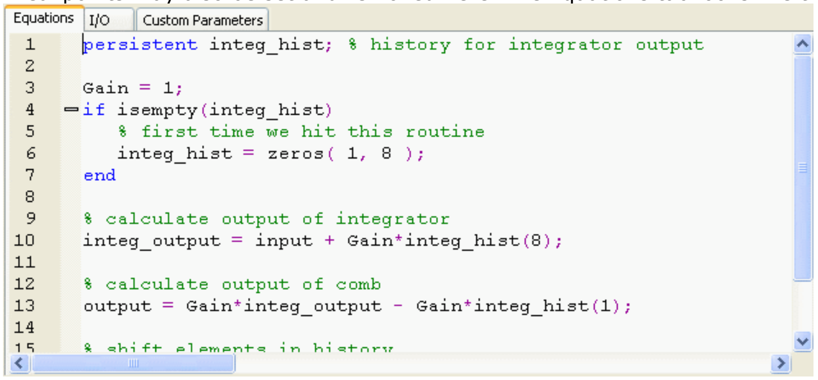
This will bring up the MathLang equations associated with the selected part in a Script Editor window, shown here:



```
persistent integ_hist; % history for integrator output

Gain = 1;
if isempty(integ_hist)
    % first time we hit this routine
    integ_hist = zeros( 1, 8 );
end

% calculate output of integrator
integ_output = input + Gain*integ_hist(8);

% calculate output of comb
output = Gain*integ_output - Gain*integ_hist(1);

% shift elements in history
integ_hist(1:7) = integ_hist(2:8);
integ_hist(8) = integ_output;
```

The Script Editor is a resizable window that supports all functions that the Equations window supports, such as setting breakpoints, Find and Replace, etc. For details about the Equations window, see *Equations* (users).

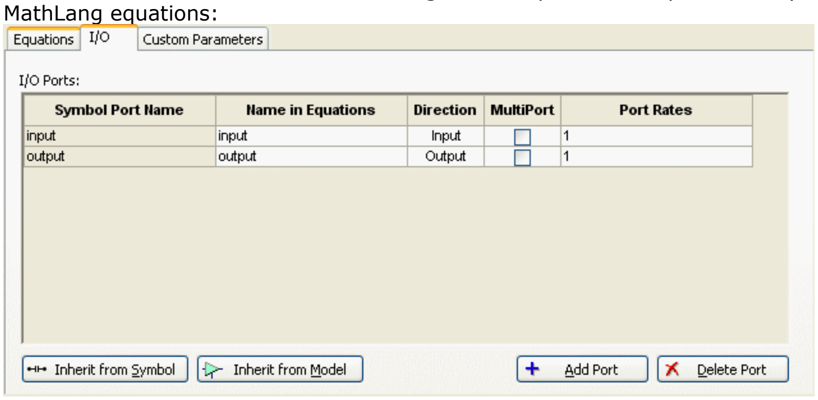
Another way to view and edit the equations is to double-click on the part in the schematic. This will bring up the standard Part Parameters dialog box with a MathLang-specific user interface. The first tab is labeled *Equations* and contains the editable equations. Breakpoints may also be set and removed here. The Equations tab looks like this:



```
persistent integ_hist; % history for integrator output

Gain = 1;
if isempty(integ_hist)
   % first time we hit this routine
   integ_hist = zeros( 1, 8 );
end

% calculate output of integrator
integ_output = input + Gain*integ_hist(8);

% calculate output of comb
output = Gain*integ_output - Gain*integ_hist(1);

% shift elements in history
```

Configuring Inputs and Outputs

The I/O tab, shown here, is used to configure the inputs and outputs used by the MathLang equations:



Symbol Port Name	Name in Equations	Direction	MultiPort	Port Rates
input	input	Input	☐	1
output	output	Output	☐	1

The first column, *Symbol Port Name*, displays the names of the ports of the current symbol assigned to the part. The current symbol can be seen in the upper-right hand corner of the Part Properties dialog box. The second column, *Name in Equations*, represents the name that the corresponding symbol port is refered to as in the MathLang equations. The *Direction* column specifies whether the corresponding symbol port is an input or an output. The *MultiPort* column specifies whether the corresponding symbol port is a bus terminal or a single terminal. **Note: Bus Terminals will produce a variable of type Cell Array, in which each element of the Cell Array corresponds to a sub-terminal of the bus.** Even if there is only one object connected to the bus, a cell array of length 1 is produced. The index into the cell array corresponds to the ordering defined in the Netlist of the part. Finally, the *Port Rates* column allows the user to set the Rate(s) of the port and is discussed in more detail in the Port Rates section below.

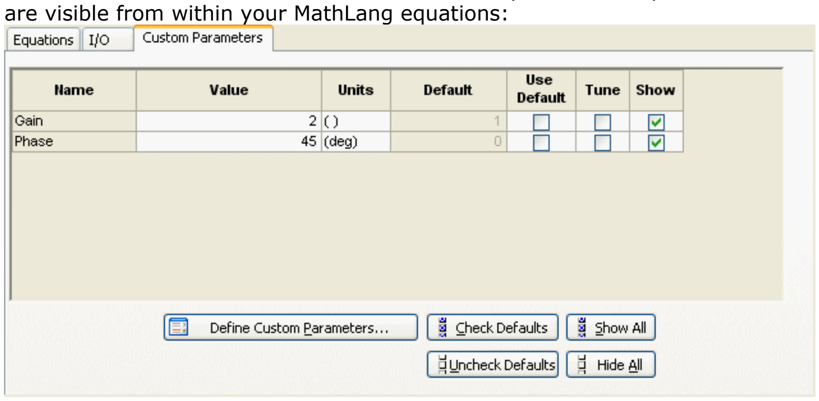
Examples referencing the above 'input' and 'output' equation names:
When the 'input' has 'Multiport' checked, but 'output' does not, then reference 'input' in the 'Equations' page using cell array notation: output = 12.5 * input { 4 }. The value assigned to 'output' is 12.5 times the 4th member of the 'input' cell array.
Similarly, when the 'output' has 'Multiport' checked, but 'input' does not, then reference 'output' in the 'Equations' page using cell array notation: output { 4 } = 12.5 * input. The value assigned to the 4th member of the 'output' cell array is 12.5 times the input.

Inputs and Outputs are added or removed by clicking the *Add Port* and *Delete Port* buttons, respectively. Note that if the *Symbol Port Name* column has an empty entry, that means the current symbol does not have a terminal to map to the port associated with that row. In other words, the port defined by that row will not receive any data.

To load preconfigured I/O configurations, *Inherit from Symbol* and *Inherit from Model* buttons are provided. Since a Symbol contains no information about directionality of a terminal, or whether the terminal is a single-port or a bus, the *Inherit from Symbol* button can only intelligently inherit symbol terminal names from a symbol. Models, on the other hand, contain all information about their ports, so the *Inherit from Model* button allows one to extract all I/O information from a model.

Custom Parameters and States

The *Custom Parameters* tab, shown here, allows you to define parameters or states that are visible from within your MathLang equations:



Name	Value	Units	Default	Use Default	Tune	Show
Gain	2	()	1	☐	☐	✓
Phase	45	(deg)	0	☐	☐	✓

The table seen in this tab allows you to set values for parameters that you have configured. This table is identical to the usual Part Parameters entry table seen in most parts. To define Parameters, use the *Define Custom Parameters* button. The resulting dialog box allows you to define parameters in a manner identical to the way one would define Parameters for a Sub-Network model (ie. the Parameters tab of a Design). See *Creating a Design with Parameters* (users) for details on defining Parameters.

Port Rates

The rate of a port determines how many samples to collect (input rate), or how many samples to generate (output rate), for each simulator clock tick.

For example, suppose we have a constant source feeding a MathLang block with the value 1 on every clock tick. If we set the input Rates parameter to be 3, the simulator will wait to collect 3 samples before firing the MathLang block's equations. Therefore, for that clock tick, the MathLang block would see a vector [1; 1; 1] as the current sample, in effect decreasing the sampling rate by a factor of 3.

Similarly, suppose that in our MathLang equations we assign a vector [1; 2; 3] to an output terminal which has an output Rate set to 3. The block would then output 3 samples: 1, 2, and 3, in effect increasing the sampling rate by a factor of 3.

Suppose input port rate is *R* (larger than 1). If input samples are with the same scalar type, the MathLang block would see a *R* x 1 vector. If the input samples are matrices with the same size *M* x *N*, the MathLang block would see a *R* x *M* x *N* structure. If input samples are in different dimensions or different sizes, the MathLang block would see a *R* x 1 cell array.

For Bus ports, the corresponding Rates parameter can be a scalar, in which case it applies that rate to all sub-terminals for that port. It may also be a vector which defines a rate for each sub-terminal separately.

How the Simulator invokes the MathLang Block

The simulator looks for 3 specific functions in the MathLang block to execute: Initialize, Run, and Finalize. **Note: If no function definition is present in your MathLang equations (ie. you just have equation script commands), then your script is wrapped into the Run function automatically.**

The Initialize function, if present, is called once at the begining of the simulation run. It can be used to pre-allocate data or perform any sort of initialization. The Finalize function, if present, is called once at the end of the simulation run and can be used to perform clean-up.

The Run function is invoked by the simulator for every clock tick, ie. every time input is presented to the block. Variables defined in the Run function do not persist between calls unless they are declared as *persistent*.

Examples

Suppose we wanted to define an Adder with a Gain parameter. First, we define configure the inputs and outputs. We define one input named "input" and check the MultiPort column since we want the adder to add an arbitrary number of inputs. We define a single output named "output" which we leave as a single-port, since the adder produces only one result.

In the Custom Parameters tab, we click the *Define Custom Parameters* button and define a single parameter called Gain.

The MathLang code which sums all of the inputs, scales by the Gain parameter, and produces an output might look like this:

```
nInputs = numel( input );
sum = 0;
for i = 1 : nInputs
 sum = sum + input{i};  % input is a cell array
end
output = Gain * sum;  % Gain was defined in Custom Parameters tab
```

Here's a more complex example that uses a persistent variable named *State* to retain state in between calls:

```
persistent State;
% initialize the structure on the first tic
if isempty( State )  % is it the first call?
 State.Amount = 1
 State.Clock  = 0
 State.Flag= 1   % increase so we don't call this block again
end
% increment to the next clock tic
State.Clock = State.Clock+1
%
% at each 100 clocks bump the amount
if ( State.Clock == 100)
State.Amount = State.Amount * 1.1
State.Clock = 0; % reset clock
end
% now scale the input data
if input >= 1
 output = State.Amount * sqrt( input )
else
 output = input
end
```

Finally, if we wanted to define Initialize and Finalize functions, we could do as follows:

```
function Initialize()
% Set up code and pre-allocation goes here
function Run()
% compute outputs based on inputs
function Finalize()
% Clean up code goes here
```

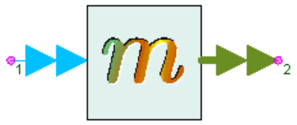
MATLAB_Cosim Part

Categories: *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
MATLAB_Cosim (algorithm)	MATLAB Cosimulation Block

MATLAB_Cosim (MATLAB Cosimulation Block)



Description: MATLAB Cosimulation Block
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *MATLAB Cosim Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
ScriptDirectory	Optional directory for location of custom Matlab files			Text	NO
MatlabSetUp	Matlab command to execute during begin method			Text	NO
MatlabFunction	Matlab command to execute for each simulation sample			Text	NO
MatlabWrapUp	Matlab command to execute during wrapup method			Text	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	multiple variant	YES

Output Ports

Port	Name	Signal Type	Optional
2	output	multiple complex matrix	NO

Notes/Equations

1. See *MATLAB Cosimulation* (sim) for detailed information about MATLAB_Cosim.

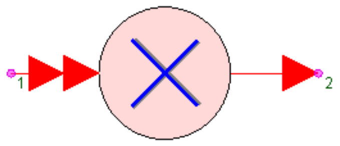
Mpy Part

Categories: *C++ Code Generation* (algorithm), *Math Matrix* (algorithm), *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
Mpy (algorithm)	Multiple Input Multiplier
MpyMultiEnv (algorithm)	Multiple Input Envelope Multiplier
MpyFxp (hardware)	Fixed Point Multiplier

Mpy(Multiple Input Multiplier)



Description: Multiple Input Multiplier
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *Mpy Part* (algorithm)

Input Ports

Port	**Name**	**Signal Type**	**Optional**
1	input	multiple anytype	NO

Output Ports

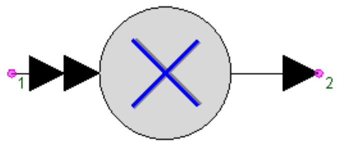
Port	**Name**	**Signal Type**	**Optional**
2	output	anytype	NO

⚠ **Note**
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim), *fixed point* (sim), or *variant* (sim).

Notes/Equations

1. Mpy outputs the product of the inputs.
2. This model reads 1 sample from all inputs and writes 1 sample to the output.
3. For a single input multiply, the input is copied to the output.

MpyMultiEnv (Multiple Input Envelope Multiplier)



Description: Multiple Input Envelope Multiplier
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Mpy Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
SideBandType	Multiplication product type: Lower sideband, Upper sideband, Both sidebands	Upper sideband		Enumeration	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	multiple envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	envelope	NO

Notes/Equations

1. This model implements a multiple input envelope multiplier.
2. When the number of inputs is 2, then this model performs exactly the same as the two input envelope multiplier *MpyEnv* (algorithm).
3. When the number of inputs is greater than 2, then the only *SideBandType* supported is *Upper sideband*. In this case, this model acts as a cascade of *MpyEnv* (algorithm) models, that is, the first two input envelope signals are multiplied using the two input *MpyEnv* (algorithm) model, the resulting signal is multiplied with the third input envelope signal using another two input *MpyEnv* (algorithm) model, the resulting signal is is multiplied with the fourth input envelope signal using another two input *MpyEnv* (algorithm) model, etc.

See:
AddEnv (algorithm)
SubEnv (algorithm)
MpyEnv (algorithm)

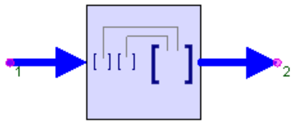
MxCom_M Part

Categories: *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
MxCom_M (algorithm)	Matrix Composer

MxCom_M (Matrix Composer)



Description: Matrix Composer
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *MxCom M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
OutputNumRows	Number of rows for output matrix	100		Integer	NO	[InputNumRows:∞)†
OutputNumColumns	Number of columns for output matrix	100		Integer	NO	[InputNumColumns:∞)‡
InputNumRows	Number of rows for input matrix	4		Integer	NO	[1:∞)
InputNumColumns	Number of columns for input matrix	4		Integer	NO	[1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	real matrix	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real matrix	NO

Notes/Equations

1. Each output matrix is composed of submatrices from the input.
2. For (OutputNumRows / InputNumRows) × (OutputNumColumns / InputNumColumns) number of input matrices, one matrix is output.
3. The output matrix is filled with submatrices in rasterized order, i.e. the top row of submatrices is filled first from left to right, and so on to the bottom row of submatrices.

See:
MxDecom_M (algorithm)
SubMx_M (algorithm)
SubMxCx_M (algorithm)

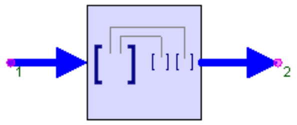
MxDecom_M Part

Categories: *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
MxDecom_M (algorithm)	Matrix Decomposer

MxDecom_M (Matrix Decomposer)



Description: Matrix Decomposer
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *MxDecom M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
StartRow	Starting row in input matrix to generate output matrices (first row is 1)	1		Integer	NO	[1:∞)
StartCol	Starting column in input matrix to generate output matrices (first column is 1, matrix upper left corner is (1,1)	1		Integer	NO	[1:∞)
InputNumRows	Number of rows for input matrix	100		Integer	NO	[OutputNumRows:∞)†
InputNumCols	Number of columns from input matrix	100		Integer	NO	[OutputNumCols:∞)‡
OutputNumRows	Number of rows for output matrix	4		Integer	NO	[1:∞)
OutputNumCols	Number of columns for output matrix	4		Integer	NO	[1:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	Input matrix to be decomposed into the output submatrices.	real matrix	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	Output matrices with dimensions OutputNumRows*OutputNumCols.	real matrix	NO

Notes/Equations

1. Each output matrix is a nonoverlapping submatrix of the input.
2. For every input matrix, (OutputNumRows / InputNumRows) × (OutputNumColumns / InputNumColumns) number of matrices are output.
3. The output matrices are extracted from the input in rasterized order, i.e. the top row of submatrices is retrieved first from left to right, and so on to the bottom row of submatrices.

See:
MxCom_M (algorithm)
SubMx_M (algorithm)
SubMxCx_M (algorithm)

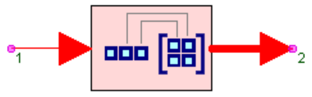
Pack_M Part

Categories: *C++ Code Generation* (algorithm), *Math Matrix* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Pack_M (algorithm)	Pack Matrix Function

Pack_M (Pack Matrix Function)



Description: Pack Matrix Function
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *Pack M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NumRows	Number of rows in output matrix	1		Integer	NO	[1:∞)
NumCols	Number of columns in output matrix	1		Integer	NO	[1:∞)
Format	Format of data to be packed into matrix: ColumnMajor, RowMajor	ColumnMajor		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype matrix	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim) or *variant* (sim).

Notes/Equations

1. The Pack_M model packs NumRows x NumCols input scalar values into an output matrix with NumRows and NumCols.
2. This model reads NumRows x NumCols scalar samples from the input and writes one matrix to the output with NumRows and NumCols.
3. Inputs are entered into the matrix output in either column-major or row-major order based using the Format parameter.

See:
Unpack_M (algorithm)
DynamicPack_M (algorithm)
DynamicUnpack_M (algorithm)

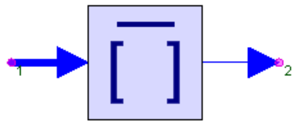
SampleMean_M Part

Categories: *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SampleMean_M (algorithm)	Matrix Mean Value

SampleMean_M (Matrix Mean Value)



Description: Matrix Mean Value
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *SampleMean M Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	input	real matrix	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. The averaged value of all elements of the input matrix is output.
2. For input matrix, one real value is output.

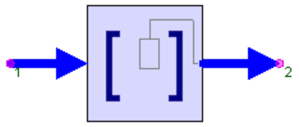
SubMx_M Part

Categories: *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
SubMx_M (algorithm)	Submatrix Extractor
SubMxCx_M (algorithm)	Complex Submatrix Extractor

SubMx_M (Submatrix Extractor)



Description: Submatrix Extractor
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *SubMx M Part* (algorithm)

Model Parameters

Name	**Description**	**Default**	**Units**	**Type**	**Runtime Tunable**	**Range**
StartRow	Starting row for the submatrix within the input matrix	1		Integer	NO	[1:∞)
StartCol	Starting column for the submatrix within the input matrix	1		Integer	NO	[1:∞)
NumRows	Number of submatrix rows	2		Integer	NO	[1:∞)
NumCols	Number of submatrix columns	2		Integer	NO	[1:∞)

Input Ports

Port	**Name**	**Signal Type**	**Optional**
1	input	real matrix	NO

Output Ports

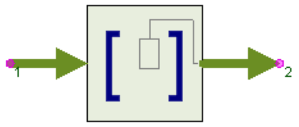
Port	**Name**	**Signal Type**	**Optional**
2	output	real matrix	NO

Notes/Equations

1. Output a submatrix of the input.
2. For every input matrix, a submatrix is output.
3. The parameters specify the dimensions of the output and the upper left position within the input where extraction begins.

See:
SubMxCx_M (algorithm)
MxDecom_M (algorithm)
MxCom_M (algorithm)

SubMxCx_M (Complex Submatrix Extractor)



Description: Complex Submatrix Extractor
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *SubMx M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
StartRow	Starting row for the submatrix within the input matrix	1		Integer	NO	[1:∞)
StartCol	Starting column for the submatrix within the input matrix	1		Integer	NO	[1:∞)
NumRows	Number of submatrix rows	1		Integer	NO	[1:∞)
NumCols	Number of submatrix columns	1		Integer	NO	[1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	complex matrix	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	complex matrix	NO

Notes/Equations

1. Output a submatrix of the input.
2. For every input matrix, a submatrix is output.
3. The parameters specify the dimensions of the output and the upper left position within the input where extraction begins.

See:
SubMx_M (algorithm)
MxDecom_M (algorithm)
MxCom_M (algorithm)

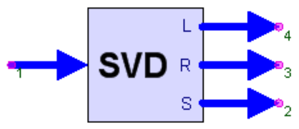
SVD_M Part

Categories: *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SVD_M (algorithm)	Singular Value Decomposition of a Toeplitz Matrix

SVD_M (Singular Value Decomposition of a Toeplitz Matrix)



Description: Singular Value Decomposition of a Toeplitz Matrix
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *SVD M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Threshold	Threshold for similarities (algorithm assumes values below Threshold have reached zero)	0.00000000000000001		Float	NO	(-∞:∞)
MaxIterations	Maximum iterations for SVD convergence	30		Integer	NO	[1:∞)
GenerateLeft	Matrix generation of left singular vectors: Do not Generate Left Singular Vectors, Generate Left Singular Vectors	Generate Left Singular Vectors		Enumeration	NO	
GenerateRight	Matrix generation of right singular vectors: Do not Generate Right Singular Vectors, Generate Right Singular Vectors	Generate Right Singular Vectors		Enumeration	NO	

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	Input stream.	real matrix	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	svals	The singular values of input - The diagonal of "W".	real matrix	NO
3	rsvec	Right singular vectors of input - "V".	real matrix	NO
4	lsvec	Left singular vectors of input - "U".	real matrix	NO

Notes/Equations

1. SVD_M computes the singular-value decomposition (SVD) of an input Toeplitz matrix A by decomposing A into A = U × W × V′, where U and V are orthogonal matrices and V′ represents the transpose of V.
2. For every input matrix, a vector is output to svals (S), a matrix is output to lsvec (L) and a matrix is output to rsvec (R).
3. The input must be a Toeplitz matrix. Output S is the diagonal of the matrix W, output L is the matrix U, and output R is the matrix V. If the input is of size M rows × N columns, S will be of size N × 1, L will be of size M × N, and V will be of size N × N.
4. The MaxIterations parameter allows the designer to limit the number of iterations that the SVD algorithm is allowed to run. This is necessary for non-convergent matrix inputs.
5. The execution time of SVD_M may be reduced by using the GenerateLeft and GenerateRight parameters to specify that the matrices of the left and right singular vectors not be generated. The vector of singular values (S) is always generated.

See:
Toeplitz_M (algorithm)

References

1. S. Haykin, *Modern Filters*, pp. 333-335, Macmillan Publishing Company, New York,

1989.

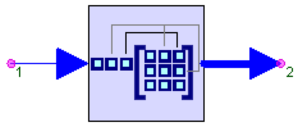
Toeplitz_M Part

Categories: *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Toeplitz_M (algorithm)	Toeplitz Matrix Converter
ToeplitzCx_M (algorithm)	Complex Toeplitz Matrix Converter

Toeplitz_M (Toeplitz Matrix Converter)



Description: Toeplitz Matrix Converter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Toeplitz M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NumRows	Number of rows in the output matrix	2		Integer	NO	[1:∞)
NumCols	Number of columns in the output matrix	2		Integer	NO	[1:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	Input stream.	real	NO

Output Ports

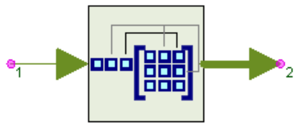
Port	Name	Description	Signal Type	Optional
2	output	the data matrix X.	real matrix	NO

Notes/Equations

1. The Toeplitz_M model packs NumRows + NumCols − 1 input real scalar values into an output real matrix with NumRows and NumCols.
2. This model reads NumRows + NumCols − 1 real scalar samples from the input and writes 1 real matrix (X) to the output with NumRows and NumCols.
3. Let M = NumCols, N = NumRows + NumCols − 1 and x[1:N] be the input, then from left to right:
   The first (top) row of X is x[M] x[M−1]) ... x[1],
   the second row of X is x[M + 1] x[M] ... x[2],
   until
   the last (bottom) row of X, which is x[N] x[N − 1] ... x[N − M + 1]. Note that N = M + NumRows − 1 and N − M + 1 = NumRows.

See:
ToeplitzCx_M (algorithm)
Identity_M (algorithm)
IdentityCx_M (algorithm)
Diagonal_M (algorithm)
DiagonalCx_M (algorithm)
Const (algorithm)

ToeplitzCx_M (Complex Toeplitz Matrix Converter)



Description: Complex Toeplitz Matrix Converter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Toeplitz M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NumRows	Number of rows in the output matrix	2		Integer	NO	[1:∞)
NumCols	Number of columns in the output matrix	2		Integer	NO	[1:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	Input stream.	complex	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	Data matrix X.	complex matrix	NO

Notes/Equations

1. The ToeplitzCx_M model packs NumRows + NumCols − 1 input complex scalar values into an output complex matrix with NumRows and NumCols.
2. This model reads NumRows + NumCols − 1 complex scalar samples from the input and writes 1 complex matrix (X) to the output with NumRows and NumCols.
3. Let M = NumCols, N = NumRows + NumCols − 1 and x[1:N] be the input, then from left to right:
   The first (top) row of X is x[M] x[M − 1]) ... x[1],
   the second row of X is x[M + 1] x[M] ... x[2],
   until
   the last (bottom) row of X, which is x[N] x[N − 1] ... x[N − M + 1]. Note that N = M + NumRows − 1 and N − M + 1 = NumRows.

See:
Toeplitz_M (algorithm)
Identity_M (algorithm)
IdentityCx_M (algorithm)
Diagonal_M (algorithm)
DiagonalCx_M (algorithm)
Const (algorithm)

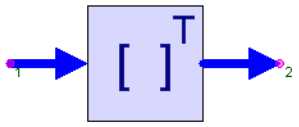
Transpose_M Part

Categories: *Math Matrix* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Transpose_M (algorithm)	Matrix Transposer
TransposeCx_M (algorithm)	Complex Matrix Transposer

Transpose_M (Matrix Transposer)



Description: Matrix Transposer
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Transpose M Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	input	real matrix	NO

Output Ports

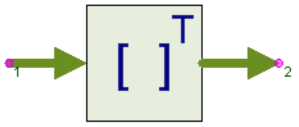
Port	Name	Signal Type	Optional
2	output	real matrix	NO

Notes/Equations

1. The transpose of the input is output.
2. If the input is a M × N matrix, then the output is a N × M matrix.
3. Let x[i,j] be an element of the input and y[j,i] be an element of the output, then y[j,i] = x[i,j].

See:
TransposeCx_M (algorithm)
Hermitian_M (algorithm)

TransposeCx_M (Complex Matrix Transposer)



Description: Complex Matrix Transposer
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Transpose M Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	input	complex matrix	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	complex matrix	NO

Notes/Equations

1. The transpose of the input is output.
2. If the input is a M × N matrix, then the output is a N × M matrix.
3. Let x[i,j] be an element of the input and y[j,i] be an element of the output, then y[j,i] = x[i,j].

See:
Transpose_M (algorithm)
Hermitian_M (algorithm)

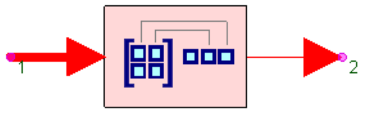
Unpack_M Part

Categories: *C++ Code Generation* (algorithm), *Math Matrix* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Unpack_M (algorithm)	Unpack Matrix Function

Unpack_M (Unpack Matrix Function)



Description: Unpack Matrix Function
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *Unpack M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NumRows	Number of rows in input matrix	1		Integer	NO	[1:∞)
NumCols	Number of columns in input matrix	1		Integer	NO	[1:∞)
Format	Format of data to be packed into matrix: ColumnMajor, RowMajor	ColumnMajor		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype matrix	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim) or *variant* (sim).

Notes/Equations

1. The Unpack_M model unpacks an input matrix to write NumRows x NumCols values to the output. If the matrix is too small, zero-padding will be used for the missing elements. If the matrix is too larger, the extra elements will be ignored.
2. This model reads one matrix from the input and writes NumRows x NumCols scalar samples to the output.
3. Elements of the input matrix of dimension, NumRow × NumCols, are output in either column-major or row-major order based using the Format parameter.

See:
Pack_M (algorithm)
DynamicPack_M (algorithm)
DynamicUnpack_M (algorithm)

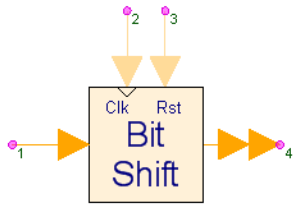
BitShiftRegister Part

Categories: *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
BitShiftRegister (algorithm)	Bit Shift Register

BitShiftRegister



Description: Bit Shift Register
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *BitShiftRegister Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NumBits	Number of bits in the output register	8		Integer	NO	[1:∞)
BitOrder	Output bit order: LSB first, MSB first	MSB first		Enumeration	YES	

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input bits	boolean	NO
2	clock	clock	int	YES
3	reset	reset	int	YES

Output Ports

Port	Name	Description	Signal Type	Optional
4	output	output bit register	boolean	NO

Notes/Equations

1. The BitShiftRegister model reads *input* bit values and writes to the *output* the current and NumBits-1 prior bits in a shift register stream with LSB or MSB first based on *BitOrder* and dependent on the state of the *reset* and *clock* inputs.
2. This model reads 1 sample from the inputs (*input* , *clock* , *reset*) and writes *NumBits* samples to the *output*. The *clock* and *reset* inputs are optional.
3. When the *clock* is not connected, it is set internally to one.
4. When the *reset* is not connected, it is set internally to zero.
5. When *reset* is one (regardless of the *clock*), then all *NumBits* samples written to the *output* are set to zero.
6. When *clock* is one, then the current *input* bit is read and is loaded into and shifts all prior bits in the *output* register.
7. The *output* resigter is sent out with either LSB (least significant bit) or MSB (most significant bit) ordering based on the setting of *BitOrder* .
8. To select only the 'n'th output bit, one can follow this part with a DownSample part with Factor set to *NumBits* and Phase set to 'n'.

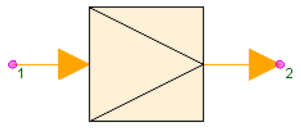
BitsToInt Part

Categories: *C++ Code Generation* (algorithm), *Math Scalar* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
BitsToInt (algorithm)	Bits to Integer Converter

BitsToInt(Bits to Integer Converter)



Description: Bits to Integer Converter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *BitsToInt Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
NumBits	Number of bits read per execution	4		Integer	NO
BitOrder	Bit order: LSB first, MSB first	MSB first		Enumeration	YES

Input Ports

Port	Name	Signal Type	Optional
1	input	boolean	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	int	NO

Notes/Equations

1. The BitsToInt model converts NumBits successive input bit values, packs these bits into an integer, and outputs the integer.
2. This model reads *NumBits* samples from the *input* and interprets it using the *BitOrder* parameter to write 1 sample to the *output*.
3. When the input is any nonzero value, a one bit value is input, otherwise a zero bit is input.
4. The *BitOrder* parameter determines how successive inputs bits are packed into the integer. If *BitOrder* is specified as *LSB first*, then the first input bit becomes the least significant bit of the output, i.e. controls whether the output is even or odd. Otherwise, the last input bit becomes the least significant bit of the output.
5. If *NumBits* is greater than the number of bits forming an integer, the most significant bits are lost.
6. If *NumBits* is greater than or equal to the number of bits forming an integer, the output could be negative. Otherwise, the output is always nonnegative.
7. When symbols are mapped onto an integer, this part becomes a BitToSymbol converter.

See:
IntToBits (algorithm)

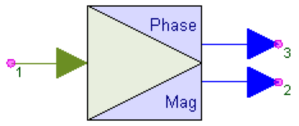
CxToPolar Part

Categories: *Math Scalar* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
CxToPolar (algorithm)	Complex to Magnitude and Phase Converter

CxToPolar (Complex to Magnitude and Phase Converter)



Description: Complex to Magnitude and Phase Converter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *CxToPolar Part* (algorithm)

Input Ports

Port	**Name**	**Signal Type**	**Optional**
1	input	complex	NO

Output Ports

Port	**Name**	**Signal Type**	**Optional**
2	magnitude	real	NO
3	phase	real	NO

Notes/Equations

1. The CxToPolar model converts input complex values to output magnitude and angle values.
2. This model reads 1 sample from the *input* and writes 1 sample to each of the outputs *magnitude* and *phase*.
   magnitude = | input |
   phase = $\angle$ input (the phase is in radians)

See:
unwrap (algorithm)
CxToRect (algorithm)
PolarToCx (algorithm)
RectToCx (algorithm)

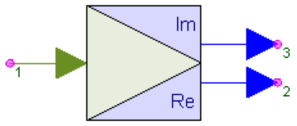
CxToRect Part

Categories: *C++ Code Generation* (algorithm), *Math Scalar* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
CxToRect (algorithm)	double CxToRect

CxToRect (Complex to Real and Imaginary Converter)



Description: double CxToRect
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *CxToRect Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	input	complex	NO

Output Ports

Port	Name	Signal Type	Optional
2	real	real	NO
3	imag	real	NO

Notes/Equations

1. The CxToRect model converts input complex values to output real and imaginary values.
2. This model reads 1 sample from the *input* and writes 1 sample to each of the outputs *real* and *imag*.
   real = real part
   imag = imaginary part

See:
CxToPolar (algorithm)
RectToCx (algorithm)
PolarToCx (algorithm)

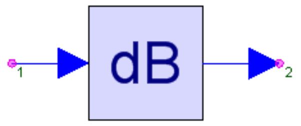
DB Part

Categories: *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
db (algorithm)	Decibel Function

db



Description: Decibel Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *DB Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Min	Minimum output value	-100		Float	YES	(-∞:∞)
DbType	Type of dB value: Power as 10*log(input), Amplitude as 20*log(input)	Amplitude as 20*log(input)		Enumeration	YES	

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. DB converts the input to a decibel (dB) scale. Zero and negative inputs convert to the Min value.
2. If the input is a power measurement set DbType to *Power as 10log(input)*, else the input is an amplitude measurement so set DbType to *Power as 20log(input)*.
3. If *Power as 10log(input)*:

$$y(n) = \begin{cases} 10\log_{10} x(n) & \text{if } 10\log_{10} x(n) \ge \text{Min} \\ \text{Min} & \text{otherwise} \end{cases}$$

If *Power as 20log(input)*:

$$y(n) = \begin{cases} 20\log_{10} x(n) & \text{if } 20\log_{10} x(n) \ge \text{Min} \\ \text{Min} & \text{otherwise} \end{cases}$$

where:
x(n) is input for sample n
y(n) is output for sample n

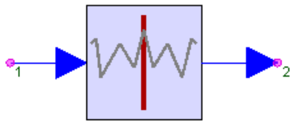
Dirichlet Part

Categories: *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Dirichlet (algorithm)	Dirichlet (Aliased Sinc) Function

Dirichlet (Dirichlet (Aliased Sinc) Function)



Description: Dirichlet (Aliased Sinc) Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Dirichlet Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
N	Length of Dirichlet kernel	10		Integer	YES	[1:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input to the Dirichlet kernel	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output of the Dirichlet kernel	real	NO

Notes/Equations

1. Dirichlet computes the normalized Dirichlet kernel (also called the aliased sinc function).
2. The value of the normalized Dirichlet kernel at x = 0 is always 1, and the normalized Dirichlet kernel oscillates between −1 and +1. The normalized Dirichlet kernel is periodic in x with a period of either 2π when N is odd or 4π when N is even.
3. The Dirichlet kernel is the discrete-time Fourier transform (DTFT) of a sampled pulse function. The parameter N is the length of the pulse [1].

See:
sinc (algorithm).

References

1. A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall: Englewood Cliffs, NJ, 1989.

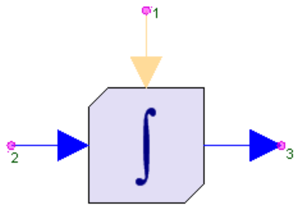
Integrator Part

Categories: *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Integrator (algorithm)	Integrator with Reset
IntegratorCx (algorithm)	Complex integrator with Reset
IntegratorInt (algorithm)	Integer Integrator with Reset

Integrator (Integrator with Reset)



Description: Integrator with Reset
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Integrator Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
IntegrationMethod	Integration method: Rectangle, Trapezoidal	Rectangle		Enumeration	YES	
LimitOutput	Output limiter options: No, Saturate, Wrap	No		Enumeration	YES	
Top	Upper limit. Visible when LimitOutput is enabled.	0		Float	YES	(-∞:∞)
Bottom	Lower limit. Visible when LimitOutput is enabled.	0		Float	YES	(-∞:∞)
InitialState	Initial integrator state	0		Float	NO	(-∞:∞)
UseIntegrationWindow	Enable integration window: No, DefinedInTime, DefinedInSamples	No		Enumeration	NO	
FeedbackGain	Gain on feedback path. Visible when UseIntegrationWindow is disabled.	1		Float	YES	(-∞:∞)
IntegrationTime	Integration time. Visible when UseIntegrationWindow is DefinedInTime.	100e-6	s	Float	NO	(0:∞)
IntegrationSamples	Integration samples. Visible when UseIntegrationWindow is DefinedInSamples.	100		Integer	NO	[2:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	reset	reset	int	YES
2	data	input	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output	real	NO

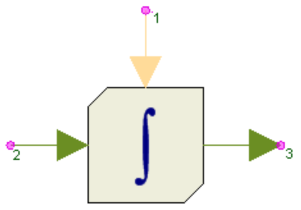
Notes/Equations

1. The input port is numerically integrated in a method chosen by the IntegrationMethod parameter. The integration value is output.
2. For every input, there is one output.
3. If the reset input is non-zero (true), the integration value is set to the current input. To enable a continuous accumulation, leave the reset port unconnected or connect a zero Const source to the reset port.
4. Limits to the output are controlled by the LimitOutput parameter. If LimitOutput is *No* , no limiting is performed, otherwise the output is kept between Top and Bottom.
5. If LimitOutput is *Saturate* then the integration value is clamped between Top and Bottom, otherwise wrap around is performed.
6. The InitialState value is applied only to the first output.

7. If UseIntegrationWindow is either *DefineInTime* or *DefineInSamples*, the integration is performed over the most recent window of samples. Samples before the first are zero valued.
8. When UseIntegrationWindow is *No*, leakage can be modeled by the FeedbackGain parameter. The output is the input plus the product of FeedbackGain and the previous integration value.
9. With the default parameters, input values are accumulated and the accumulation is output.

See:
IntegratorCx (algorithm)
IntegratorInt (algorithm)
SlidWinAvg (algorithm)

IntegratorCx



Description: Complex integrator with Reset
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Integrator Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
IntegrationMethod	Integration method: Rectangle, Trapezoidal	Rectangle		Enumeration	YES	
LimitOutput	Output limiter options: No, Saturate, Wrap	No		Enumeration	YES	
Top	Upper magnitude limit. Visible when LimitOutput is enabled.	0		Float	YES	(-∞:∞)
Bottom	Lower magnitude limit. Visible when LimitOutput is enabled.	0		Float	YES	(-∞:∞)
InitialState	Initial integrator state	0		Complex number	NO	(-∞:∞)
UseIntegrationWindow	Enable integration window: No, DefinedInTime, DefinedInSamples	No		Enumeration	NO	
FeedbackGain	Gain on feedback path	1		Float	YES	(-∞:∞)
IntegrationTime	Integration time. Visible when UseIntegrationWindow is DefinedInTime.	100e-6	s	Float	NO	(0:∞)
IntegrationSamples	Integration samples. Visible when UseIntegrationWindow is DefinedInSamples.	100		Integer	NO	[2:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	reset	reset	int	YES
2	data	input	complex	NO

Output Ports

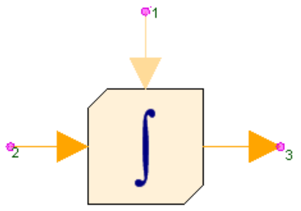
Port	Name	Description	Signal Type	Optional
3	output	output	complex	NO

Notes/Equations

1. The input port is numerically integrated in a method chosen by the IntegrationMethod parameter. The integration value is output.
2. For every input, there is one output.
3. If the reset input is non-zero (true), the integration value is set to the current input. To enable a continuous accumulation, leave the reset port unconnected or connect a zero Const source to the reset port.
4. Limits to the output are controlled by the LimitOutput parameter. If LimitOutput is *No* , no limiting is performed, otherwise the output magnitude is kept between Top and Bottom.
5. If LimitOutput is *Saturate* then the magnitude of the integration value is clamped between Top and Bottom, otherwise wrap around is performed.
6. The InitialState value is applied only to the first output.
7. If UseIntegrationWindow is either *DefineInTime* or *DefineInSamples*, the integration is performed over the most recent window of samples. Samples before the first are zero valued.
8. When UseIntegrationWindow is *No*, leakage can be modeled by the FeedbackGain parameter. The output is the input plus the product of FeedbackGain and the previous integration value.
9. With the default parameters, input values are accumulated and the accumulation is output.

See also: *Integrator* (algorithm), *IntegratorInt* (algorithm), *SlidWinAvg* (algorithm).

IntegratorInt



Description: Integer Integrator with Reset
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Integrator Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
LimitOutput	Output limiter options: No, Saturate, Wrap	No		Enumeration	YES	
Top	Upper integer limit. Visible when LimitOutput is enabled.	0		Integer	YES	(-∞:∞)
Bottom	Lower integer limit. Visible when LimitOutput is enabled.	0		Integer	YES	(-∞:∞)
InitialState	Initial integrator state	0		Integer	NO	(-∞:∞)
UseIntegrationWindow	Enable integration window: No, DefinedInTime, DefinedInSamples	No		Enumeration	NO	
IntegrationTime	Integration time. Visible when UseIntegrationWindow is DefinedInTime.	100e-6	s	Float	NO	(0:∞)
IntegrationSamples	Integration samples. Visible when UseIntegrationWindow is DefinedInSamples.	100		Integer	NO	[2:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	reset	reset	int	YES
2	data	input	int	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output	int	NO

Notes/Equations

1. Input is summed and output.
2. For every input, there is one output.
3. If the reset input is non-zero (true), the sum is set to the current input. To enable a continuous accumulation, leave the reset port unconnected or connect a zero Const source to the reset port.
4. Limits to the output are controlled by the LimitOutput parameter. If LimitOutput is *No* , no limiting is performed, otherwise the output is kept between Top and Bottom.
5. If LimitOutput is *Saturate* then the sum is clamped between Top and Bottom, otherwise wrap around is performed.
6. The InitialState value is applied only to the first output.
7. If UseIntegrationWindow is either *DefineInTime* or *DefineInSamples*, the integration is performed over the most recent window of samples. Samples before the first are zero valued.
8. With the default parameters, input values are accumulated and the accumulation is output.

See also: *Integrator* (algorithm), *IntegratorCx* (algorithm), *SlidWinAvg* (algorithm).

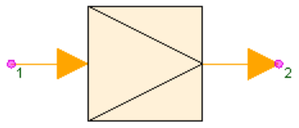
IntToBits Part

Categories: *C++ Code Generation* (algorithm), *Math Scalar* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
IntToBits (algorithm)	Integer to Bits Converter

IntToBits (Integer to Bits Converter)



Description: Integer to Bits Converter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *IntToBits Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
NumBits	Number of output bits per input sample	4		Integer	NO
BitOrder	Bit order: LSB first, MSB first	MSB first		Enumeration	YES

Input Ports

Port	Name	Signal Type	Optional
1	input	int	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	boolean	NO

Notes/Equations

1. The IntToBits model extracts the least significant *NumBits* number of bits from an integer input and serial outputs the bits as zeros and ones.
2. This model reads 1 sample from the *input* and interprets it using the *BitOrder* parameter to write *NumBits* samples to the *output*.
3. If BitOrder is specified as *LSB first*, then the least significant bit is output first. If nBits is greater than the number of bits forming an integer, the most significant bits above the number of bits forming an integer are lost and are assigned the input sign bit, i.e. a one bit if the input is negative.
4. When symbols are mapped onto an integer, this part becomes a SymbolToBits converter.
5. To select only the 'n'th output bit, one can follow this part with a DownSample part with Factor set to *NumBits* and Phase set to 'n'.

See:
BitsToInt (algorithm)

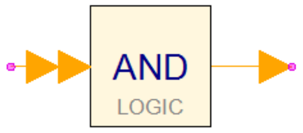
Logic Part

Categories: *C++ Code Generation* (algorithm), *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Logic (algorithm)	Boolean Logic Function
AND_Fxp (hardware)	Fixed Point Bitwise AND
NAND_Fxp (hardware)	Fixed Point Bitwise NAND
NOR_Fxp (hardware)	Fixed Point Bitwise NOR
NOT_Fxp (hardware)	Fixed Point NOT
OR_Fxp (hardware)	Fixed Point Bitwise OR
XNOR_Fxp (hardware)	Fixed Point Bitwise XNOR
XOR_Fxp (hardware)	Fixed Point Bitwise XOR

Logic (Boolean Logic Function)



Description: Boolean Logic Function
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Logic Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Logic	Logic operation: NOT, AND, NAND, OR, NOR, XOR, XNOR	AND		Enumeration	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	Input logic values.	multiple boolean	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	Result of the logic test, with FALSE equal to zero and TRUE equal to a non-zero integer (not necessarily 1).	boolean	NO

Notes/Equations

1. A logical operation selected by the Logic parameter is applied to the inputs. The *NOT* operation can have only one input.
2. The inputs are interpreted as booleans, i.e. false when the input is zero, otherwise true.
3. This part's symbol text changes when the "Logic" parameter value is changed. That is "AND", "OR", "XOR", etc. is based on the value of the "Logic" parameter. This is accomplished via a %ParameterName% macro (in the annotation text), a feature which is end-user accessable. See *Symbols* (users) for more info.

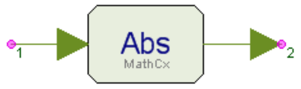
MathCx Part

Categories: *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
MathCx (algorithm)	Complex Math Function

MathCx (Complex Math Function)



Description: Complex Math Function
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Math Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
FunctionType	Mathematical function: Abs, Ceil, Exp, Floor, Ln, Log10, Pow10, Recip, Round, Sqr, Sqrt, Conj	Abs		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	complex	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	complex	NO

Notes/Equations

1. Math calculates complex mathematical functions:
   y(n) = f(x(n))
   where:
   f( ) is a function selected by the FunctionType parameter
   x(n) is input for sample n
   y(n) is output for sample n
2. If the function cannot be calculated, the simulation is halted with an error message.
3. For FunctionType:

- *Abs*, y(n) = |x(n)| = $\sqrt{Re\{x(n)\}^2 + Im\{x(n)\}^2}$
- *Ceil*, y(n) = $\lceil Re\{x(n)\}\rceil + j \times \lceil Im\{x(n)\}\rceil$
  - See *Ceil* function of Math component.
- *Exp*, y(n) = $e^{x(n)} = e^{Re\{x(n)\}}$ (cos(Im{x(n)}) + j sin(Im{x(n)}))
- *Floor*, y(n) = $\lfloor Re\{x(n)\}\rfloor + j \times \lfloor Im\{x(n)\}\rfloor$
  - See *Floor* function of Math component.
- *Ln*, y(n) = $\ln(x(n)) = \ln(|x(n)|) + j \times \angle x(n)$, where $\angle x(n)$ is the phase of x(n) in radians.
- *Log10*, y(n) = $\log_{10}$ (x(n)) = ln(x(n)) / ln(10)
- *Pow10*, y(n) = $10^{x(n)} = e^{x(n)\ln(10)}$
- *Recip*, y(n) = 1 / x(n) = (Re{x(n)} − j Im{x(n)}) / $|x(n)|^2$
- *Round*, y(n) = *Round*(Re{x(n)}) + j *Round*(Im{x(n)})
  - See *Round* function of Math component.
- *Sqr*, y(n) = $x(n)^2$
- *Sqrt*, y(n) = $\sqrt{x(n)} = \sqrt{|x(n)|} \times e^{j \times 0.5 \times \angle x(n)}$, where $\angle x(n)$ is the phase of x(n) in radians.
- *Conj*, y(n) = $Re\{x(n)\} - j\ \ Im\{x(n)\}$

See:
Math (algorithm)
Trig (algorithm)
TrigCx (algorithm)

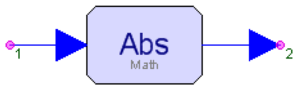
Math Part

Categories: *C++ Code Generation* (algorithm), *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Math (algorithm)	Math Function
MathCx (algorithm)	Complex Math Function
ABS_Fxp (hardware)	Fixed Point Absolute Value
SqrtFxp (hardware)	Fixed Point Square Root

Math (Math Function)



Description: Math Function
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Math Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
FunctionType	Mathematical function: Abs, Ceil, Exp, Floor, Ln, Log10, Pow10, Recip, Round, Sqr, Sqrt, Sgn	Abs		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. Math calculates real mathematical functions:
   y(n) = f(x(n))
   where:
   f( ) is a function selected by the FunctionType
   x(n) is input for sample n
   y(n) is output for sample n
2. If the function cannot be calculated, the simulation is halted with an error message.
3. For FunctionType:
   - *Abs*, y(n) = |x(n)|
   - *Ceil*, y(n) = $\lceil x(n) \rceil$, where $x(n) \le \lceil x(n) \rceil < x(n) + 1$
   - *Exp*, y(n) = e $^{x(n)}$
   - *Floor*, y(n) = $\lfloor x(n) \rfloor$, where $x(n) - 1 < \lfloor x(n) \rfloor \le x(n)$
   - *Ln*, y(n) = ln(x(n))
   - *Log10*, y(n) = log $_{10}$ (x(n))
   - *Pow10*, y(n) = 10 $^{x(n)}$
   - *Recip*, y(n) = 1 / x(n)
   - *Round*, y(n) = closest integer to x(n)
     - When x(n) is at the same distance between integers, Round maps away from 0, e.g. 2.5 maps to 3 and −2.5 maps to −3.
   - *Sqr*, y(n) = x(n) ^{2}
   - *Sqrt*, y(n) = $\sqrt{x(n)}$
   - *Sgn*, y(n) = a number indicating the sign of the input n i.e., y(n) = 1 if n > 0; y(n) = 0 if n == 0; and y(n) = -1 if n < 0.

See:
MathCx (algorithm)
Trig (algorithm)
TrigCx (algorithm)

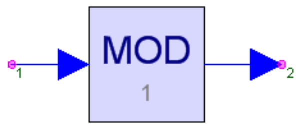
Modulo Part

Categories: *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Modulo (algorithm)	Modulo Function
ModuloInt (algorithm)	Integer Modulo Function

Modulo (Modulo Function)



Description: Modulo Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Modulo Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Modulo	Modulo value	1		Float	YES	(-∞:0) or (0:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	real	NO

Output Ports

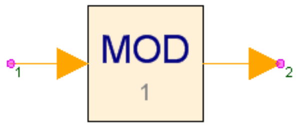
Port	Name	Description	Signal Type	Optional
2	output	output signal	real	NO

Notes/Equations

1. Modulo outputs the remainder with the same sign as input after dividing the input by the Modulo parameter.
   $y(n) = \text{fmod}\, x(n)$
   where:
   x(n) is input for sample n
   y(n) is output for sample n
2. For every input, there is an output.

See:
ModuloInt (algorithm)

ModuloInt



Description: Integer Modulo Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Modulo Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Modulo	Integer modulo value	8		Integer	YES	(0:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	int	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	int	NO

Notes/Equations

1. ModuloInt functions identically to *Modulo* (algorithm) except that the operation is over integers.

See:
Modulo (algorithm)

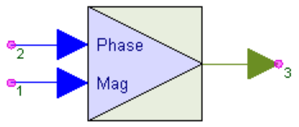
PolarToCx Part

Categories: *Math Scalar* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
PolarToCx (algorithm)	Magnitude and Phase to Complex Converter

PolarToCx (Magnitude and Phase to Complex Converter)



Description: Magnitude and Phase to Complex Converter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *PolarToCx Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	magnitude	real	NO
2	phase	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	output	complex	NO

Notes/Equations

1. The PolarToCx model converts input magnitude and angle values to output complex values.
   output = *magnitude* × (cos(*phase*) + j × sin(*phase*))
2. This model reads 1 sample from the inputs *magnitude* and *phase* and writes 1 sample to the *output*.
3. The phase angle is specified in radians.

See:
PolarToRect (algorithm)
CxToPolar (algorithm)
RectToPolar (algorithm)

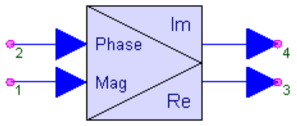
PolarToRect Part

Categories: *Math Scalar* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
PolarToRect (algorithm)	Magnitude and Phase to Rectangular Converter

PolarToRect (Magnitude and Phase to Rectangular Converter)



Description: Magnitude and Phase to Rectangular Converter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *PolarToRect Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	magnitude	real	NO
2	phase	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	x	real	NO
4	y	real	NO

Notes/Equations

1. The PolarToCx model converts input magnitude and phase values to output complex values.
   - *x* = *magnitude* * cos(*phase*)
   - *y* = *magnitude* * sin(*phase*)
2. This model reads 1 sample from the inputs *magnitude* and *phase* and writes 1 sample to the outputs *x* and *y*.
3. The phase is specified in radians.

See:
PolarToCx (algorithm)
RectToPolar (algorithm)
CxToPolar (algorithm)

Polynomial Part

Categories: *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Polynomial (algorithm)	Polynomial Mapper
PolynomialInt (algorithm)	Integer Polynomial Mapper

Polynomial (Polynomial Mapper)

Description: Polynomial Mapper
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Polynomial Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Coefficients	Polynomial coefficients (0-th order coefficient first)	[0, 1]		Floating point array	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	real	NO

Notes/Equations

1. Polynomial models a system with a polynomial input-output relationship.
2. For every input, there is an output.
3. If the input is x, the output is $y = c_0 + c_1 \times x + c_2 \times x^2 + \ldots + c_N \times x^N$, where N is the order of the polynomial and $c_0, \ldots, c_N$ are the elements of the Coefficients parameter.

See:
PolynomialInt (algorithm)

PolynomialInt

Description: Integer Polynomial Mapper
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Polynomial Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Coefficients	Integer polynomial coefficients (0-th order coefficient first)	[0, 1]		Integer array	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	int	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	int	NO

Notes/Equations

1. PolynomialInt functions identically to *Polynomial* (algorithm) except that the operation is over integers.

See:
Polynomial (algorithm)

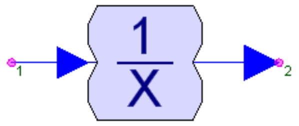
Reciprocal Part

Categories: *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Reciprocal (algorithm)	Reciprocal Function

Reciprocal (Reciprocal Function)



Description: Reciprocal Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Reciprocal Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
MagLimit	Magnitude limit; non-zero value limits the output magnitude	0		Float	YES	(-∞:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. The reciprocal of the input is calculated with an optional magnitude limit.
   If MagLimit = 0
   $$y(n) = \frac{1}{x(n)}$$
   If MagLimit ≠ 0 and input = 0
   y(n) = MagLimit
   If MagLimit ≠ 0 and input ≠ 0
   $$y(n) = \begin{cases} \text{MagLimit} & \text{if } \frac{1}{x(n)} > \text{MagLimit} \\ -\text{MagLimit} & \text{if } \frac{1}{x(n)} < -\text{MagLimit} \\ \frac{1}{x(n)} & \text{otherwise} \end{cases}$$
   where:
   x(n) is input for sample n
   y(n) is output for sample n

See:
Math (algorithm)
MathCx (algorithm)

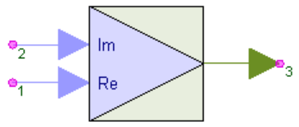
RectToCx Part

Categories: *C++ Code Generation* (algorithm), *Math Scalar* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
RectToCx (algorithm)	double RectToCx

RectToCx (Real and Imaginary to Complex Converter)



Description: double RectToCx
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *RectToCx Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	real	real	YES
2	imag	real	YES

Output Ports

Port	Name	Signal Type	Optional
3	output	complex	NO

Notes/Equations

1. The RectToCx model converts input real and imaginary values to output complex values.
   - *output* = *real* + j × *imaginary*
2. This model reads 1 sample from the inputs *real* and *imaginary* and writes 1 sample to the *output*.

See:
RectToPolar (algorithm)
CxToRect (algorithm)
PolarToRect (algorithm)

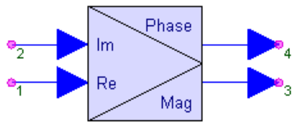
RectToPolar Part

Categories: *Math Scalar* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
RectToPolar (algorithm)	Rectangular to Polar Converter

RectToPolar (Rectangular to Polar Converter)



Description: Rectangular to Polar Converter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *RectToPolar Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	x	real	NO
2	y	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	magnitude	real	NO
4	phase	real	NO

Notes/Equations

1. The RectToPolar model converts input real and imaginary values to output magnitude and phase values.
   - *magnitude* = sqrt(*real*^2 + *imaginary*^2)
   - *phase* = atan2(*imaginary*, *real*)
2. This model reads 1 sample from the inputs *real* and *imaginary* and writes 1 sample to the outputs *magnitude* and *phase*.
3. The phase angle is in the range -π to π radians.

See:
RectToCx (algorithm)
PolarToRect (algorithm)
CxToRect (algorithm)

Rotate Part

Categories: *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Rotate (algorithm)	Complex Rotate Function

Rotate

Description: Complex Rotate Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Rotate Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
RotationAngle	Rotation angle	0	deg	Float	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	complex	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	complex	NO

Notes/Equations

1. Rotate rotates a complex input.
2. For every input, there is an output.
3. The input is multiplied by exp(j × θ) where θ is the RotationAngle parameter value in radians.

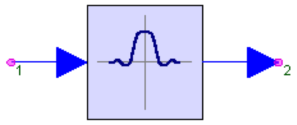
Sinc Part

Categories: *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
sinc (algorithm)	Sinc Function

sinc



Description: Sinc Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Sinc Part* (algorithm)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	The input x to the sinc function.	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	The output of the sinc function.	real	NO

Notes/Equations

1. The sinc function is defined as sin(x)/x which is continuous with value 1.0 when x = 0. Input is in radians.
2. For every input, there is an output.

See:
Dirichlet (algorithm)

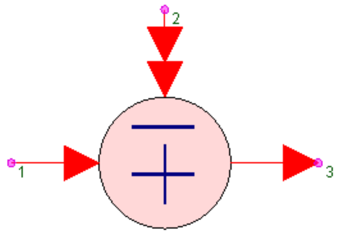
Sub Part

Categories: *C++ Code Generation* (algorithm), *Math Matrix* (algorithm), *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Sub (algorithm)	Multiple Input Subtractor
SubEnv (algorithm)	Multiple Input Subtractor
SubFxp (hardware)	Fixed Point Subtractor

Sub (Multiple Input Subtractor)



Description: Multiple Input Subtractor
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *Sub Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	pos	anytype	NO
2	neg	multiple anytype	NO

Output Ports

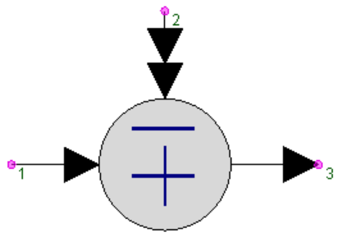
Port	Name	Signal Type	Optional
3	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim), *fixed point* (sim), or *variant* (sim).

Notes/Equations

1. The Sub model subtracts all inputs at the *neg* input from the input at the *pos* input and produces the result at the output.
2. This model reads 1 sample from all inputs and writes 1 sample to the output.
3. For discussion on the variant type, see *Variant* (sim).

SubEnv (Multiple Input Subtractor)



Description: Multiple Input Subtractor
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Sub Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
OutputFc	Output characterization frequency for the combined signal: Min, Max, Center, User defined	Center		Enumeration	NO		O
UserDefinedFc	User defined output characterization frequency	100e6	Hz	Float	NO	[0:∞)†	F

Input Ports

Port	Name	Description	Signal Type	Optional
1	pos		envelope	NO
2	neg	input signals	multiple envelope	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output signal	envelope	NO

Notes/Equations

1. The SubEnv model subtracts all envelope signals connected to the *neg* port from the envelope signal connected to the *pos* port.
2. This model reads 1 sample from all inputs and writes 1 sample to the output.
3. For more information see the documentation of the *AddEnv* (algorithm) model.

See:
AddEnv (algorithm)
MpyEnv (algorithm)
MpyMultiEnv (algorithm)

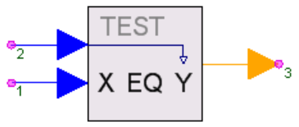
Test Part

Categories: *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Test (algorithm)	Comparison Test Function
CompareFxp (hardware)	Fixed Point Compare

Test (Comparison Test Function)



Description: Comparison Test Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Test Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Condition	Test condition: EQ, NE, GT, GE, LT, LE	EQ		Enumeration	NO	
Tolerance	Finite-precision comparison tolerance	0		Float	YES	[0:∞)
CrossingDetection	Crossing detection options: Off, Positive, Negative, All	Off		Enumeration	YES	
InitialCondition	Initial condition value for crossing detection: FALSE, TRUE	FALSE		Enumeration	NO	

Input Ports

Port	Name	Description	Signal Type	Optional
1	input1	Signal used in left hand side of comparison test	real	NO
2	input2	Signal used in right hand side of comparison test	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	Test result	int	NO

Notes/Equations

1. The Test model compares the value of the *input1* input to the value of the *input2* input using the relationship selected by the *Condition* parameter. The output result of the comparison is either FALSE or TRUE. FALSE is represented by a value of 0 and TRUE is represented by a value of 1.
2. For one input at *input1* and *input2*, there is one output.
3. If the *CrossingDetection* parameter is set to *Off*, then *output* is just the result of the comparison as described below:
   - If *Condition* is *EQ*, then *output* is TRUE if $|\ input2\ -\ input1\ | \le Tolerance$, else *output* is FALSE
   - If *Condition* is *NE*, then *output* is TRUE if $|\ input2\ -\ input1\ | > Tolerance$, else *output* is FALSE
   - If *Condition* is *GT*, then *output* is TRUE if $input1 > input2$, else *output* is FALSE
   - If *Condition* is *GE*, then *output* is TRUE if $input1 \ge input2$, else *output* is FALSE
   - If *Condition* is *LT*, then *output* is TRUE if $input1 < input2$, else *output* is FALSE
   - If *Condition* is *LE*, then *output* is TRUE if $input1 \le input2$, else *output* is FALSE
4. If the *CrossingDetection* parameter is set to *Positive*, then *output* is TRUE when the comparison result changes from FALSE (previous comparison value) to TRUE (current comparison value), else *output* is FALSE.
5. If the *CrossingDetection* parameter is set to *Negative*, then *output* is TRUE when the comparison result changes from TRUE (previous comparison value) to FALSE (current comparison value), else *output* is FALSE.
6. If the *CrossingDetection* parameter is set to *All*, then *output* is TRUE when the comparison result changes from its previous value, else *output* is FALSE.
7. When the *CrossingDetection* parameter is set to *Positive*, *Negative*, or *All*, an initial

value is needed to compare the result of the first comparison against. This value is set in the *InitialCondition* parameter.

Trig Part

Categories: *C++ Code Generation* (algorithm), *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Trig (algorithm)	Trigonometric Function
TrigCx (algorithm)	Complex Trigonometric Function

Trig (Trigonometric Function)

Description: Trigonometric Function
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Trig Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
FunctionType	Trigonometric function type: Sin, Cos, Tan, Cot, Asin, Acos, Atan, Acot, Sinh, Cosh, Tanh, Coth, Asinh, Acosh, Atanh, Acoth	Sin		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. Trig calculates real trigonometric functions:
   y(n) = f(x(n))
   where:
   f( ) is a function selected by the Type parameter
   x(n) is input for sample n
   y(n) is output for sample n
2. If the function cannot be calculated, the simulation is halted with an error message.
3. All angles are in radians.

See:
TrigCx (algorithm)
Math (algorithm)
MathCx (algorithm)

TrigCx (Complex Trigonometric Function)

1
Sin
TrigCx
2

Description: Complex Trigonometric Function
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Trig Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
FunctionType	Complex trigonometric function type: Sin, Cos, Tan, Cot, Asin, Acos, Atan, Acot, Sinh, Cosh, Tanh, Coth, Asinh, Acosh, Atanh, Acoth	Sin		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	complex	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	complex	NO

Notes/Equations

1. TrigCx calculates complex trigonometric functions:
   y(n) = f(x(n))
   where:
   f( ) is a function selected by the Type parameter
   x(n) is input for sample n
   y(n) is output for sample n
2. The principal value is calculated.
3. If the function cannot be calculated, the simulation is halted with an error message.
4. All angles are in radians.

See:
Trig (algorithm)
Math (algorithm)
MathCx (algorithm)

References

1. Handbook of Mathematical Functions, 1972, Abramowitz and Stegun.

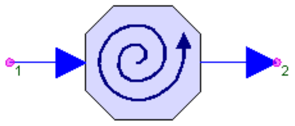
Unwrap Part

Categories: *Math Scalar* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
unwrap (algorithm)	Unwrap Phase

unwrap



Description: Unwrap Phase
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Unwrap Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
OutPhase	Initial output phase	0	deg	Float	YES	(-∞:∞)
PrevPhase	Initial wrapped phase of input signal for computing the first phase difference	0	deg	Float	YES	(-∞:∞)
PhaseType	Phase type for the input and output signals: radians, degrees	radians		Enumeration	YES	

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. Unwrap a phase plot by removing discontinuities of magnitude 2 π.
2. For every input, there is an output.
3. The phase is assumed to never change by more than π between inputs. The input is assumed to be in the range $[-\pi, \pi]$.

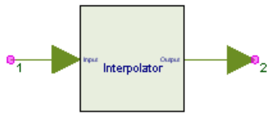
Interpolator Part

Categories: *C++ Code Generation* (algorithm), *OFDM* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Interpolator (algorithm)	1-dimension Wiener interpolator

Interpolator



Description: 1-dimension Wiener interpolator
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Interpolator Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
R	interpolation ratio	2		Integer	NO
L	half the original sample number used for interpolation	4		Integer	NO
Alpha	ratio of signal bandwidth to half the sample rate [0,1]	0.5		Float	NO
SNR	signal to noise power ratio of input signal in dB	100		Float	NO
OperationMode	Operation mode (in Continuous mode, the filtering delay is R*L samples): Blocked, Continuous	Blocked		Enumeration	NO
BlockSize	Block size of input signal (> 2*L)	128		Integer	NO
OutputOption	Normal: InBlockSize x R, Tailored: InBlockSize x R - R + 1: Normal, Tailored	Tailored		Enumeration	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	Input	signal to be interpolated	complex	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Output	signal interpolated	complex	NO

Notes/Eqautions

1. The interpolator implements a 1-dimension wiener low-pass interpolation for baseband signal.
2. The interpolator assumes the input signal has a square spectrum mask within the digital frequency span [0,Alpha], where Alpha is the the ratio of signal bandwidth in Hz to half the sampling rate in Hz. It also assumes the noise is AWGN type.
3. SNR is the Signal to Noise power ratio in dB which is used to optimize the filter coefficient so as to suppress out-of-band noise in input signal.
4. OptiontionMode specifies the operation mode of the interpolator.
    - If OperationMode==Continuous, the interpolator will consumes 1 sample from the Input each Run and export R samples to the Output. There's a delay of R*L samples in this mode, i.e. the (R*L+1)'th output sample shall be synchrounous with the 1'th input sample, and the (R*L+R+1)'th output synchronize the 2'th output, etc.
    - If Opeationmode==Blocked, the interpolator will consumes BlockSize samples from the Input each Run.
        - If OutputOption is Normal, it will export R*BlockSize samples to the Output, with the first output sample synchronized to the first input sample.
        - If OutputOption is Tailored, it will export R*BlockSize-(R-1) samples to the Output, with the first output sample synchronized to the first input sample and the last of output synchronized to the last of input. This case is applicable to channel estimation in OFDM receiver.

References

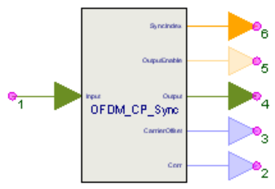
OFDM_CP_Sync Part

Categories: *C++ Code Generation* (algorithm), *OFDM* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
OFDM_CP_Sync (algorithm)	OFDM cyclic prefix/postfix based synchronization

OFDM_CP_Sync



Description: OFDM cyclic prefix/postfix based synchronization
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *OFDM CP Sync Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
DFTSize	DFT Size before oversampling	2048		Integer	NO
CPSize	Cyclic prefix/postfix Size before oversampling	512		Integer	NO
OperationMode	Operation mode: Continuous, Burst	Continuous		Enumeration	NO
BurstLength	Burst frame length before oversampling	8192		Integer	NO
OversampleRatio	Oversampling ratio: x1, x2, x4, x8, x16, x32, x64, x128	x1		Enumeration	NO
PeakThreshold	Correlation coefficient threshold for peak search	0.7		Float	NO
TimingTolerance	Non oversampled DFT window jitter tolerance	5		Integer	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	Input	Input signal	complex	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Corr	Correlation magnitude of input and local PN samples	real	NO
3	CarrierOffset	Normalized carrier offset ((Fc(Rx)-Fc(Tx)*(Ts*DFTSize))	real	NO
4	Output	Synchronized output signal aligned with integeral times of FrameSize (carrier offset not compensated)	complex	NO
5	OutputEnable	Synchronization validation flag of output signal	int	NO
6	SyncIndex	Number of input samples from real frame start to ideally aligned frame start	int	NO

Notes/Eqautions

1. This model completes the timing synchronization based on CP correlation. It can detect the correct timing position and estimate the carrier offset. It can provide the correlation values, aligned output, output enable indicator, timing offset values and carrier offset.
2. x1, x2, ... x128 of OversampleRatio means the actual sampling ratio. All parameters specified in this model are non-oversample values, the actual number of these parameters still need to multiple $2^{\mathrm{OversampleRatio}}$. For instance, DFTSizeOS=DFTSize*$2^{\mathrm{OversampleRatio}}$.
3. In OFDM symbol, the cyclic prefix is the same as the corresponding samples in the rear of the DFT samples. So the correlation of the two parts can indicate the symbol timing position because there will be a correlation peak to indicate the timing position.
   When the input signals from Input pin are Input(n), the relative correlation values corr(n) is outputted from Corr output pin.

$$Corr(n) = \frac{\left| \sum_{i=0}^{CPSizeOS-1} Input(n-i) \cdot Input^*(n-i-DFTSizeOS) \right|}{\frac{1}{2} \cdot \left(\sum_{i=0}^{CPSizeOS-1} \left| Input(n-i) \right|^2 + \sum_{i=0}^{CPSizeOS-1} \left| Input(n-i-DFTSizeOS) \right|^2 \right)}$$

4. This model can work in two modes: Continuous and Burst, which can be specified in OperationMode.
5. In Continuous mode, this model calculates the correlation for each sample. It begins with the acquiring phase. In this phase, it searches the correlation peak. The correlation peak value must be larger than the parameter PeakThreshold. When it finds the correlation peak, it enters the tracking phase. The correlation peak position is just the last sample of the OFDM symbol. In tracking phase, this model keeps searching the correlation peaks for the following OFDM symbols. If the first peak appears at sample index i, the following correlation peaks are expected at i+(DFTSizeOS+CPSizeOS), i+2*(DFTSizeOS+CPSizeOS), i+3*(DFTSizeOS+CPSizeOS),…. In tracking phase, it will detect how many samples the following peaks position bias from the expectation positions. If the bias is not larger than TimingToleranceOS=TimingTolerance*$2^{OversampleRatio}$, it will keep the timing synchronization position that is detected at the first time; otherwise, it will lose the synchronization and enter the acquiring phase again.

   When it enters the tracking phase, it can align the signals of Output pin to the integral times of SymbolLengthOS (SymbolLengthOS=DFTSizeOS+CPSizeOS), which will be convenient to deal with the signals in the following models. For example, if it finds the correlation peak at sample index i, [Input(i-SymbolLengthOS+1), Input(i-SymbolLengthOS+2), ... ,Input(i)] is an OFDM symbol. If i is within sample index [n*SymbolLengthOS, n*SymbolLengthOS+SymbolLengthOS/2], it will output this OFDM symbol from sample index (n+1)*SymbolLengthOS.

   Output((n+1)*SymbolLengthOS)=Input(i-SymbolLengthOS+1)

   Output((n+1)*SymbolLengthOS+1)= Input(i-SymbolLengthOS+2)

   ......

   If i is within sample index (n*SymbolLengthOS+SymbolLengthOS/2,(n+1)*SymbolLengthOS], it will output this OFDM symbol from sample index (n+2)*SymbolLengthOS.

   Output((n+2)*SymbolLengthOS)= Input(i-SymbolLengthOS+1)

   Output((n+2)*SymbolLengthOS+1)= Input(i-SymbolLengthOS+2)

   ......

   If it doesn’t lose synchronization, it will keep the relationship between Input and Output signals.

   When it aligns the Output to the integral times of SymbolLengthOS, OutputEnable pin will output 1 to indicate the corresponding signals from Output pin are aligned.

   The SyncIndex pin will also output the oversampled sample number that need to adjust to align the input signals to the corresponding Output pin signals.

   The CarrierOffset pin will output $\Delta f \bullet T_u$ that is detected from corresponding outputted symbol, where Δf is the actual carrier offset frequency and T_u=DFTSize•T, T is the non-oversample sampling period.
6. In Burst mode, a parameter BurstLength can be specified, which is a non-oversample value. Then in a BurstLengthOS (BurstLengthOS=BurstLength*$2^{OversampleRatio}$) interval, only one OFDM symbol can be detected based on CP correlation. When it finds the correlation peak, it will stop the correlation calculation until the end of this burst. Then in the next burst, it searches the correlation peak again and get the new synchronization position.

   The Output pin signals need to align to the integral times of BurstLengthOS. For example, if it finds the correlation peak at sample index i, Input(i-SymbolLengthOS+1) is the first sample of the burst. If i is within sample index [n*BurstLengthOS, (n+1)*BurstLengthOS-SymbolLengthOS/2], it will output this burst from sample index (n+1)*BurstLengthOS.

   Output((n+1)*BurstLengthOS)=Input(i-SymbolLengthOS+1)

Output((n+1)*BurstLengthOS+1)= Input(i-SymbolLengthOS+2)

......

Output((n+1)*BurstLengthOS+BurstLengthOS-1)=Input(i-SymbolLengthOS+BurstLengthOS)

If i is within sample index ((n+1)*BurstLengthOS-SymbolLengthOS/2,(n+1)*BurstLengthOS], it will output this OFDM symbol from sample index (n+2)*BurstLengthOS.

Output((n+2)*BurstLengthOS)= Input(i-SymbolLengthOS+1)

Output((n+2)*BurstLengthOS+1)= Input(i-SymbolLengthOS+2)

......

Output((n+2)*BurstLengthOS+BurstLengthOS-1)= Input(i-SymbolLengthOS+BurstLengthOS)

For the next burst, it will search the peak position again and align Output pin signals according to the new peak position.

The OutputEnable, SyncIndex and CarrierOffset pins will output the signals for the corresponding burst, which are similar to those of Continuous mode.

References

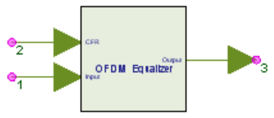
OFDM_Equalizer Part

Categories: *C++ Code Generation* (algorithm), *OFDM* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
OFDM_Equalizer (algorithm)	OFDM single tap equalizer

OFDM_Equalizer



Description: OFDM single tap equalizer
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *OFDM Equalizer Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
MagnitudeClip	Clip signal magnitude when channel gain is very low: NO, YES	NO		Enumeration	NO
MaxMagnitude	Maximum magnitude for clipping	sqrt(2)*2		Float	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	Input	Input signal	complex	NO
2	CFR	Channel frequency response	complex	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	Output	Equalized signal	complex	NO

Notes/Eqautions

1. This model does the one-tap equalization for OFDM signals.
2. The received sub-carriers signals can be connect to “Input” pin and the corresponding channel frequency responses can be connect to “CFR” pin. Then the equalized signals can be outputted.
3. If MagnitudeClip is NO, Output = Input/CFR. If MagnitudeClip is YES, then the magnitude of Output will be clipped to MaxMagnitude.
4. If CFR inputs are zeros, the corresponding output signals will also be zeros.

Refrences

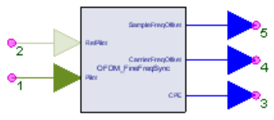
OFDM_FineFreqSync Part

Categories: *C++ Code Generation* (algorithm), *OFDM* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
OFDM_FineFreqSync (algorithm)	OFDM RF carrier frequency offset and ADC sampling frequency offset estimation

OFDM_FineFreqSync



Description: OFDM RF carrier frequency offset and ADC sampling frequency offset estimation
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *OFDM FineFreqSync Part* (algorithm)

Model Parameters

Name	**Description**	**Default**	**Units**	**Type**	**Runtime Tunable**
DFTSize	DFT size before oversampling	64		Integer	NO
GuardSize	Guard interval size before oversampling	16		Integer	NO
NumPilot	Pilot number in OFDM symbol	4		Integer	NO
PilotIndexDim	Dimension of parameter PilotIndex: Dimension_1, Dimension_2	Dimension_1		Enumeration	NO
PilotIndex	Pilot position in OFDM symbol [-FFTSize/2, FFTSize/2-1]	[-21,-7,7,21]		Integer array	NO
SymbolInterval	OFDM symbol index difference be Pilot and RefPilot ([1,128])	1		Integer	NO

Input Ports

Port	**Name**	**Description**	**Signal Type**	**Optional**
1	Pilot	Non-equalized pilot carriers picked up from OFDM, from negative to positive	complex	NO
2	RefPilot	Non-equalized pilot carriers picked up from Previous OFDM symbol	complex	YES

Output Ports

Port	**Name**	**Description**	**Signal Type**	**Optional**
3	CPE	Common phase error between Pilot and RefPilot	real	NO
4	CarrierFreqOffset	Normalized RF carrier frequency offset,(Fc(rx)-Fc(tx))*(Ts*DFTSize)	real	NO
5	SampleFreqOffset	Normalized ADC sampling frequency offset,(Fs(rx)-Fs(tx))/Fs(tx)	real	NO

Notes/Equations

1. This model estimates the normalized RF carrier frequency offset Δf0, the normalized ADC sampling clock frequency offset ΔF0 and the Common Phase Error (CPE) using Least Linear Squares Estimator (LLSE) algorithm.

   $$\Delta f0 = (f0_{RX} - f0_{TX}) * (T_s * N_{DFT})$$

   where $f0_{RX}$ and $f0_{TX}$ are the RF carrier frequencies in Hz in the receiver side and the transmitter side respectively, T_s is the sample rate of the based OFDM symbol in Sec and N_{DFT} is the corresponding DFT size. $(T_s * N_{DFT})$ represents the reciprocal of sub-carrier space.

   $$\Delta F0 = (F0_{RX} - F0_{TX}) / F0_{TX}$$

   where $F0_{RX}$ and $F0_{TX}$ are the ADC sampling clock frequency in the receiver side and the transmitter side respectively.
2. Non-equalized pilots from the same subcarrier positions of two OFDM symbols are used. SymbolInterval specifies the difference of OFDM symbol index between the Pilot and RefPilot. If RefPilot is disconnected, the model regard the delayed version of

Pilot as reference.

3. Assuming $\mathbf{\theta} = [\Delta f0, \Delta F0]^T$,

   p_i is the index of the ith pilot in OFDM symbol,

   Δf_{pi} = angle(Pilot[i]*RefPilot[i]) * $N_{DFT}/(N_{DFT}+N_{Guard})$ / SymbolInterval / (2π), is the phase difference of the ith pilot of Pilot and RefPilot,

   $\mathbf{v} = [\Delta f_{p1}, \Delta f_{p2}, \Delta f_{p3}, \ldots, \Delta f_{pK}]$,

   $\mathbf{u} = [1, 1, 1, \ldots, 1]^T$, $\mathbf{p} = [p_1, p_2, p_3, \ldots, p_K]^T$, where K is the number of continuous pilots in one OFDM symbol,

   $\mathbf{H} = [\mathbf{u}, \mathbf{p}]$,

   We can get the estimation equations

   $\hat{\mathbf{\theta}} = \mathbf{A}^T * \mathbf{v}$, where $\mathbf{A} = (\mathbf{H}^T\mathbf{H})^{-1} * \mathbf{H}^T$,

   and

   $CPE = 2\pi * \Delta f0 * T_s * (N_{DFT}+N_{Guard}) * SymbolInterval$.

References

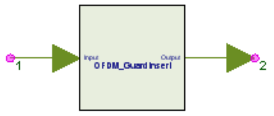
OFDM_GuardInsert Part

Categories: *C++ Code Generation* (algorithm), *OFDM* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
OFDM_GuardInsert (algorithm)	OFDM guard interval insertion

OFDM_GuardInsert



Description: OFDM guard interval insertion
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *OFDM GuardInsert Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
DFTSize	DFT size before oversampling	64		Integer	NO
PrefixSize	Prefix size(s) before oversampling	[16]		Integer array	NO
PostfixSize	Postfix size(s) before oversampling	[0]		Integer array	NO
GuardStuff	Signals to be stuffed in guard interval: CyclicShift, Zeros	CyclicShift		Enumeration	NO
OversampleRatio	Oversampling ratio: x1, x2, x4, x8, x16, x32, x64, x128	x1		Enumeration	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	Input	Input signal	complex	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Output	Output signal	complex	NO

Notes/Eqautions

1. This model inserts guard intervals to OFDM symbols. The input are consecutive OFDM time-domain signals from IDFT (Inverser Digital Fourier Transformation) module.
2. Both prefix and postfix can be added to input signal.
3. The stuff signal may be cyclic shift (extension) of an IDFT period or zeros.
4. Different guard intervals may be added to different OFDM symbols. For instance, in a WLAN 11a signal with 4 data symbols, the input symbols is

   [STS, STS, LTS, LTS, SIGNAL, Data, Data, Data, Data],

   the parameters may be set as

   DFTSize = 64,

   PrefixSize = 16*[1,1,2,0,1,1,1,1,1],

   PostfixSize = zeros(9, 1),

   GuardStuff = CyclicShift.

Refrences

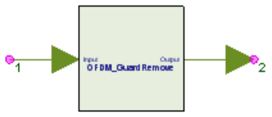
OFDM_GuardRemove Part

Categories: *C++ Code Generation* (algorithm), *OFDM* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
OFDM_GuardRemove (algorithm)	OFDM guard interval removal

OFDM_GuardRemove



Description: OFDM guard interval removal
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *OFDM GuardRemove Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
DFTSize	DFT size before oversampling	64		Integer	NO
PrefixSize	Prefix size(s) before oversampling	[16]		Integer array	NO
PostfixSize	Postfix size(s) before oversampling	[0]		Integer array	NO
GuardStuff	Signals stuffed in guard interval: CyclicShift, Zeros	CyclicShift		Enumeration	NO
CIRLength	Channel impulse response length for cyclic overlap addition (the first CIRLength*2^OversampleRatio samples should be chopped before applying guard removal)	8		Integer	NO
CIRAdjust	Number of non-oversampled samples to cyclic delay (shift from left to right) channel impulse response	0		Integer	NO
OversampleRatio	Oversampling ratio: x1, x2, x4, x8, x16, x32, x64, x128	x1		Enumeration	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	Input	Input signal	complex	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Output	Output signal	complex	NO

Notes/Eqautions

1. This model removes the guard interval of the OFDM symbol. It outputs OFDM time-domain signals to DFT (Digital Fourier Transformation) module directly.
2. It can remove different guard intervals for different OFDM symbols. For instance, in a WLAN 11a signal with 4 data symbols, the input symbols is

   [STS, STS, LTS, LTS, SIGNAL, Data, Data, Data, Data],

   the parameters may be set as

   DFTSize = 64,

   PrefixSize = 16*[1,1,2,0,1,1,1,1,1],

   PostfixSize = zeros(9, 1),

   GuardStuff = CyclicShift.

3. x1, x2, ... x128 of OversampleRatio means the actual sampling ratio. All parameters specified in this model are all non-oversample values, such as DFTSize, PrefixSize, PostfixSize, CIRLength and CIRAdjust. The actual numbers of oversampled samples for these parameters are the product of these non-oversample parameters and $2^{OversampleRatio}$. For instance, DFTSizeOS=DFTSize*$2^{OversampleRatio}$.

4. It can work in two modes: CyclicShift and Zeros for cyclic guard interval and zero padding guard interval removal. The mode can be specified in parameter GuardStuff.
5. In CyclicShift mode, it can remove cyclic guard interval. Parameter CIRAdjust can be specified to get the DFT signals with cyclic delay, which can shift the CIR (channel impulse response) in the time domain. For example, PrefixSize is [pre1,pre2]; PostfixSize is [post1,post2]. Each run, there are two OFDM symbols in the Input pin. The two OFDM symbols have 2*DFTSizeOS+sum(PrefixSizeOS)+sum(PostfixSizeOS) samples. It will output 2*DFTSizeOS samples of DFT signals. The input sequence of the first OFDM symbol is

   $[Input_{-pre1OS}, \dots, Input_{-1}, Input_{0}, \dots, Input_{DFTSizeOS-1}, Input_{DFTSizeOS}, \dots, Input_{DFTSizeOS+post1OS-1}]$

   where $[Input_{-pre1OS}, \dots, Input_{-1}]$ is prefix, $[Input_{0}, \dots, Input_{DFTSizeOS-1}]$ is DFT window samples, $[Input_{DFTSizeOS}, \dots, Input_{DFTSizeOS+post1OS-1}]$ is postfix.

   The output sequence is

   $[Input_{mod(-CIRAdjustOS+0,DFTSizeOS)}, Input_{mod(-CIRAdjustOS+1,DFTSizeOS)}, \dots, Input_{mod(-CIRAdjustOS+DFTSizeOS-1,DFTSizeOS)}]$.

   The second OFDM symbol is similar as the first one.
6. In Zeros mode, it can remove zero padding guard interval. Parameter CIRLength is the non-oversampled samples number covered by maximum multipath delay. In zero padding mode, it needs to add the CIRLengthOS samples that follow DFT window to the first CIRLengthOS samples in DFT window. So the input begins with the CIRLengthOSth sample and ends in the CIRLengthOSth sample of next OFDM symbol. For example, PrefixSize is [pre1,pre2]; PostfixSize is [post1,post2]. Each run, there are two OFDM symbols in the Input pin. The two OFDM symbols have 2*DFTSizeOS+sum(PrefixSizeOS)+sum(PostfixSizeOS) samples. It will output 2*DFTSizeOS samples of DFT signals. The input sequence of the first OFDM symbol is

   $[Input_{-pre1OS+CIRLengthOS}, \dots, Input_{-1}, Input_{0}, \dots, Input_{DFTSizeOS-1}, Input_{DFTSizeOS}, \dots, Input_{DFTSizeOS+post1OS-1}, Input_{DFTSizeOS+post1OS}, \dots, Input_{DFTSizeOS+post1OS+CIRLengthOS-1}]$

   where $[Input_{DFTSizeOS+post1OS}, \dots, Input_{DFTSizeOS+post1OS+CIRLengthOS-1}]$ is actually the prefix of the next OFDM symbol.

   It will add CIRLengthOS samples that follow the DFT window to the first CIRLengthOS samples in DFT window to get the DFTSizeOS samples:

   $Y_{0\sim DFTSizeOS-1}=[Input_{0}+Input_{DFTSizeOS}, \dots, Input_{CIRLengthOS-1}+Input_{DFTSizeOS+CIRLengthOS-1}, Input_{CIRLengthOS}, \dots, Input_{DFTSizeOS-1}]$

   Then the sequence can be cyclically delayed by CIRAdjust to get the output sequence:

   $Output_{i}=Y_{mod(-CIRLengthOS+i,DFTSizeOS)}$

   where $i=0\sim DFTSizeOS-1$

   The second OFDM symbol is similar as the first one.

Refrences

OFDM_SubcarrierDemux Part

Categories: *C++ Code Generation* (algorithm), *OFDM* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
OFDM_SubcarrierDemux (algorithm)	OFDM subcarrier demultiplexing

OFDM_SubcarrierDemux

1
Input
Ouput
OFDM_SubcarrierDemux
2

Description: OFDM subcarrier demultiplexing
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *OFDM SubcarrierDemux Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
DFTSize	DFT size before oversampling	64		Integer	NO
NumOutput	Number of output branches ([1,4])	1		Integer	NO
Out1_NumCarriers	Number of (branch 1) Subcarriers in each OFDM symbol	[48]		Integer array	NO
Out1_DimCarrierIndex	Dimension of parameter Out1_CarrierIndex: 1-D, 2-D, 1-D_FromFile	2-D		Enumeration	NO
Out1_FileFormat	Specify the format of the Out1 IndexFile;In ASCII format, data are seperated by space or changing lines: ASCII, Binary	ASCII		Enumeration	NO
Out1_CarrierIndex	Out1 subcarrier position array (1-D, list the indices one by one; 2-D, list the index zone by 2-digit pairs)	[-26,-22; -20,-8; -6,-1; 1,6; 8,20; 22,26]		Integer array	NO
Out1_FileName	Out1 index file specification			Filename	NO
Out2_NumCarriers	Number of (branch 2) Subcarriers in each OFDM symbol	[4]		Integer array	NO
Out2_DimCarrierIndex	Dimension of parameter Out2_CarrierIndex: 1-D, 2-D, 1-D_FromFile	1-D		Enumeration	NO
Out2_FileFormat	Specify the format of the Out2 IndexFile;In ASCII format, data are seperated by space or changing lines: ASCII, Binary	ASCII		Enumeration	NO
Out2_CarrierIndex	Out2 subcarrier position array (1-D, list the indices one by one; 2-D, list the index zone by 2-digit pairs)	[-21,-7,7,21]		Integer array	NO
Out2_FileName	Out2 index file specification			Filename	NO
Out3_NumCarriers	Number of (branch 3) Subcarriers in each OFDM symbol	[4]		Integer array	NO
Out3_DimCarrierIndex	Dimension of parameter Out3_CarrierIndex: 1-D, 2-D, 1-D_FromFile	1-D		Enumeration	NO
Out3_FileFormat	Specify the format of the Out3 IndexFile;In ASCII format, data are seperated by space or changing lines: ASCII, Binary	ASCII		Enumeration	NO
Out3_CarrierIndex	Out3 subcarrier position array (1-D, list the indices one by one; 2-D, list the index zone by 2-digit pairs)	[-21,-7,7,21]		Integer array	NO
Out3_FileName	Out3 index file specification			Filename	NO
Out4_NumCarriers	Number of (branch 4) Subcarriers in each OFDM symbol	[4]		Integer array	NO
Out4_DimCarrierIndex	Dimension of parameter Out4_CarrierIndex: 1-D, 2-D, 1-D_FromFile	1-D		Enumeration	NO
Out4_FileFormat	Specify the format of the Out4 IndexFile;In ASCII format, data are seperated by space or changing lines: ASCII, Binary	ASCII		Enumeration	NO
Out4_CarrierIndex	Out4 subcarrier position array (1-D, list the indices one by one; 2-D, list the index zone by 2-digit pairs)	[-21,-7,7,21]		Integer array	NO
Out4_FileName	Out4 index file specification			Filename	NO
InputOrder	Subcarrier input order ([0~DFTSize/2-1, -DFTSize/2~-1] or [-DFTSize/2~-1, 0~DFTSize/2-1]), not applicable to EVMRef: DC_Pos_Neg, Neg_DC_Pos	DC_Pos_Neg		Enumeration	NO
OversampleRatio	Oversampling ratio: x1, x2, x4, x8, x16, x32, x64, x128	x1		Enumeration	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	Input	OFDM frequency domain signals from DFT	complex	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	Output	Signals demultiplexed from input subcarriers	multiple complex	NO

Notes/Equations

1. This model de-multiplexes different type of signals from OFDM symbols in frequency-domain. The input signal is from DFT module.
2. The parameters have similar meaning with that of OFDM_SubcarrierMux and should be set accordingly. See *OFDM_SubcarrierMux* (algorithm) for details.

References

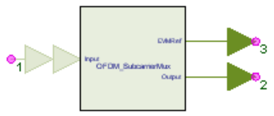
OFDM_SubcarrierMux Part

Categories: *C++ Code Generation* (algorithm), *OFDM* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
OFDM_SubcarrierMux (algorithm)	OFDM subcarrier multiplexing

OFDM_SubcarrierMux



Description: OFDM subcarrier multiplexing
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *OFDM SubcarrierMux Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
DFTSize	DFT size before oversampling	64		Integer	NO
NumInput	Number of input branches ([1,4])	1		Integer	NO
In1_NumCarriers	Number of (branch 1) Subcarriers in each OFDM symbol	[48]		Integer array	NO
In1_DimCarrierIndex	Dimension of parameter In1_CarrierIndex: 1-D, 2-D, 1-D_FromFile	2-D		Enumeration	NO
In1_FileFormat	Specify the format of the In1 IndexFile;In ASCII format, data are seperated by space or changing lines: ASCII, Binary	ASCII		Enumeration	NO
In1_CarrierIndex	In1 subcarrier position array (1-D, list the indices one by one; 2-D, list the index zone by 2-digit pairs)	[-26,-22; -20,-8; -6,-1; 1,6; 8,20; 22,26]		Integer array	NO
In1_FileName	In1 index file specification			Filename	NO
In2_NumCarriers	Number of (branch 2) Subcarriers in each OFDM symbol	[4]		Integer array	NO
In2_DimCarrierIndex	Dimension of parameter In2_CarrierIndex: 1-D, 2-D, 1-D_FromFile	1-D		Enumeration	NO
In2_FileFormat	Specify the format of the In2 IndexFile;In ASCII format, data are seperated by space or changing lines: ASCII, Binary	ASCII		Enumeration	NO
In2_CarrierIndex	In2 subcarrier position array (1-D, list the indices one by one; 2-D, list the index zone by 2-digit pairs)	[-21,-7,7,21]		Integer array	NO
In2_FileName	In2 index file specification			Filename	NO
In3_NumCarriers	Number of (branch 3) Subcarriers in each OFDM symbol	[4]		Integer array	NO
In3_DimCarrierIndex	Dimension of parameter In3_CarrierIndex: 1-D, 2-D, 1-D_FromFile	1-D		Enumeration	NO
In3_FileFormat	Specify the format of the In3 IndexFile;In ASCII format, data are seperated by space or changing lines: ASCII, Binary	ASCII		Enumeration	NO
In3_CarrierIndex	In3 subcarrier position array (1-D, list the indices one by one; 2-D, list the index zone by 2-digit pairs)	[-21,-7,7,21]		Integer array	NO
In3_FileName	In3 index file specification			Filename	NO
In4_NumCarriers	Number of (branch 4) Subcarriers in each OFDM symbol	[4]		Integer array	NO
In4_DimCarrierIndex	Dimension of parameter In4_CarrierIndex: 1-D, 2-D, 1-D_FromFile	1-D		Enumeration	NO
In4_FileFormat	Specify the format of the In4 IndexFile;In ASCII format, data are seperated by space or changing lines: ASCII, Binary	ASCII		Enumeration	NO
In4_CarrierIndex	In4 subcarrier position array (1-D, list the indices one by one; 2-D, list the index zone by 2-digit pairs)	[-21,-7,7,21]		Integer array	NO
In4_FileName	In4 index file specification			Filename	NO
OutputOrder	Subcarrier output order ([0~DFTSize/2-1, -DFTSize/2~-1] or [-DFTSize/2~-1, 0~DFTSize/2-1]), not applicable to EVMRef: DC_Pos_Neg, Neg_DC_Pos	DC_Pos_Neg		Enumeration	NO
OversampleRatio	Oversampling ratio: x1, x2, x4, x8, x16, x32, x64, x128	x1		Enumeration	NO
CustomEVMRef	Custom EVM reference or not: NO, YES	NO		Enumeration	NO
EVMRef_NumCarriers	Number of EVM reference subcarriers in each OFDM symbol	[52]		Integer array	NO
EVMRef_DimCarrierIndex	Dimension of parameter EVMRef_CarrierIndex: 1-D, 2-D, 1-D_FromFile	2-D		Enumeration	NO
EVMRef_FileFormat	Specify the format of the EVMRefFile;In ASCII format, data are seperated by space or changing lines: ASCII, Binary	ASCII		Enumeration	NO
EVMRef_CarrierIndex	EVMRef subcarrier position array with each row a continuous range	[-26,-1; 1,26]		Integer array	NO
EVMRef_FileName	EVMRef index file specification			Filename	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	Input	Signals to be placed in subcarriers	multiple complex	YES

Output Ports

Port	Name	Description	Signal Type	Optional
2	Output	OFDM frequency domain format ready for applying IDFT	complex	NO
3	EVMRef	EVM reference subcarriers	complex	NO

Notes/Equations

1. This model multiplexes different types of signals in frequency-domain to corresponding subcarrier locations. The Output is ready to be processed by the Inverse Digital Fourier Transformation (IDFT).
2. Each branch of the input bus may be data of an OFDMA user or pilots.
3. For each of the 4 branches supported, the parameters are configured as described below (<N> has one of the values 1, 2, 3, 4)

   In<N>_NumCarriers specifies the number of subcarriers in each OFDM symbol. For instance, in DVB-T signal, there are 142 SP (Scattered Pilot) in each OFDM symbol and they repeat every 4 OFDM symbols, so this parameter should be [142, 142, 142, 142].

   In<N>_DimCarrierIndex specifies the form of parameter In<N>_CarrierIndex. 1-D is suitable for non-consecutive subcarrier allocation whereas 2-D is suitable for consecutive subcarrier allocation.

   In<N>_CarrierIndex should be a 1-D or 2-D array whose elements belong to the interval [-DFTSize/2, DFTSize/2-1]. A negative index indicates that the subcarrier will be modulated to the left half of the RF spectrum, a 0 index indicates that the subcarrier will be modulated at the center of the RF spectrum, and a positive index indicates that the subcarrier will be modulated to the right half of the RF spectrum

   RF spectrum: [f_{min}, ..., **fc**, ..., f_{max}]

   Carrier index: [$-K_{min}$, ..., -1, **0**, 1, ..., K_{max}], K_i<DFTSize/2

   If In<N>_DimCarrierIndex is 1-D, the parameter In<N>_CarrierIndex should be a 1-D array of sum(In<N>_NumCarriers) elements. The first In<N>_NumCarriers(1) elements specify the subcarrier location in the first OFDM symbol, the next NumCarriers(2) elements specify the subcarrier location in the second OFDM symbol, etc. For SP in DVB-T signal, it should be [-852, -840, ..., 828, 840, -849, -837, ..., 831, 843, -846, -834, ..., 834, 846, -843, -831, ..., 837, 849].

   If In<N>_DimCarrierIndex is 2-D, the parameter In<N>_CarrierIndex should be a 2-column 2-D array. Each line in this array specifies a consecutive subcarrier zone. In DVB-T signal, the location of Data subcarriers are [-851, -850, ..., -841, -839, -838, ..., -830, -829, -827 ...] in the form of 1-D array, and the corresponding 2-D array should be [-851,-841; -839,-829;-827,...].
4. If there is subcarrier overlap in input branches, for instance, both branch i and branch j have allocation to the -kth subcarrier, the data in the max(i,j)th branch will be the final value.
5. OutputOrder is used to set the subcarrier order of the Output pin. If OutputOrder is DC_Pos_Neg, i.e. [Direct Current, Positive, Negative], the output subcarrier indices will be [0, 1, ..., DFTSizeOS/2-1, -DFTSizeOS/2, -DFTSizeOS/2+1, ..., -1], which matches the formula of IDFT, where DFTSizeOS is DFTSize*$2^{OversampleRatio}$. If OutoutOrder is Neg_DC_Pos, the output subcarrier indices will be [-DFTSizeOS/2, -DFTSizeOS/2+1, ..., -1, 0, 1, ..., DFTSizeOS/2-1], which matches the subcarrier allocation in RF spectrum.
6. x1, x2, ... x128 of OversampleRatio means DFTSize*$2^{OversampleRatio}$-DFTSize zero subcarriers will be inserted to the Output signal, which will ease the upsampling before applying ADC or digital IF up-conversion.
7. This model also provides a pin as the reference signal of EVM (Error Vector Magnitude) measurement. If CustomEVMRef is NO, the output will be the all the non-zero sub-carriers from the input branches in the order from Negative frequency to Positive frequency. If CustomEVMRef is YES, the custom EVMRef output can be set manually. Note that the size of EVMRef_NumCarriers must be a factor of the LCM (Least Common Multiple) of (size(In1_NumCarriers), size(In2_NumCarriers), ...).
8. In each execution of this model, there will be DFTSize*$2^{OversampleRatio}$ *LCM(size(In1_NumCarriers), size(In2_NumCarriers), ...) samples in the Output pin. The corresponding numbers of input samples are sum(In1_NumCarriers)*LCM(size(In1_NumCarriers), size(In2_NumCarriers), ...)/size(In1_NumCarriers), sum(In2_NumCarriers)*LCM(size(In1_NumCarriers),

size(In2_NumCarriers), ...)/size(In2_NumCarriers), etc.

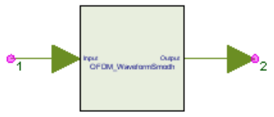
OFDM_WaveformSmooth Part

Categories: *C++ Code Generation* (algorithm), *OFDM* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
OFDM_WaveformSmooth (algorithm)	OFDM waveform smoothing

OFDM_WaveformSmooth



Description: OFDM waveform smoothing
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *OFDM WaveformSmooth Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
DFTSize	DFT size(s) before oversampling	[64]		Integer array	NO
PrefixSize	Prefix size(s) before oversampling	[16]		Integer array	NO
PostfixSize	Postfix size(s) before oversampling	[0]		Integer array	NO
RCSlopeLength	RC slope length before oversampling [0,PrefixSize+PostfixSize]	0		Float	NO
OversampleRatio	Oversampling ratio: x1, x2, x4, x8, x16, x32, x64, x128	x1		Enumeration	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	Input	Input signal	complex	NO

Output Ports

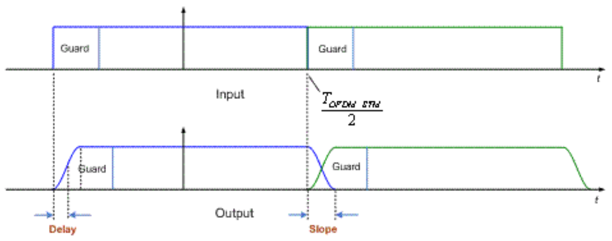
Port	Name	Description	Signal Type	Optional
2	Output	Output signal	complex	NO

Notes/Equations

1. This model smoothes the OFDM signal with cyclic prefix/postfix with Raised Cosine (RC) slope. It's only applicable for OFDM symbols with prefix/postfix as guard intervals.
2. RCSlopeLength is set in the form of a multiple of non-oversampled sample interval, integer or fraction.
3. The RC slope is added by cyclicly extending each OFDM symbol (the cyclic period is the OFDM symbol duration without guard interval) and then applying a Raised Cosine window to each as follows:

$$w_T(t)=\begin{cases}\sin^2\left(\dfrac{\pi}{2}\left(0.5+\dfrac{t}{T_{TR}}\right)\right) & -\dfrac{T_{TR}}{2}<t<\dfrac{T_{TR}}{2}\\ 1 & \dfrac{T_{TR}}{2}\le t<T-\dfrac{T_{TR}}{2}\\ \sin^2\left(\dfrac{\pi}{2}\left(0.5-\dfrac{t-T}{T_{TR}}\right)\right) & T-\dfrac{T_{TR}}{2}\le t<T+\dfrac{T_{TR}}{2}\end{cases}$$

   where T is the OFDM symbol duration with guard interval, T_{TR} is the transient (Slope in the figure below) duration.
4. There will be a delay of ceil(RCSlopeLength*$2^{OversampleRatio}$/2) samples in the output signal.

4. 

5. In general, the higher the OversampleRatio is, the more smooth the waveform is. The longer the RC slope is, the lower the out-of-band power is. Nevertheless, long RC slope will reduce the multi-path fading and timing error immunity in the receiver side.

References

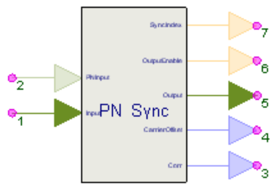
PN_Sync Part

Categories: *C++ Code Generation* (algorithm), *OFDM* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
PN_Sync (algorithm)	Pseudo noise sequence based synchronization

PN_Sync



Description: Pseudo noise sequence based synchronization
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *PN Sync Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
FrameSize	Frame Size without oversampling	1024		Integer	NO
OversampleRatio	Oversampling ratio: x1, x2, x4, x8, x16, x32, x64, x128	x1		Enumeration	NO
PNCodeSampPerSym	Number of samples per local PN code symbol: x1, x2, x4, x8	x1		Enumeration	NO
PNCodeLength	Number of PN code symbols in each PN code pattern	128		Integer	NO
PNCodePatternNum	Number of PN code patterns	1		Integer	NO
PNCode	PN code sequence	[1+0i]		Complex array	NO
RCRolloffFactor	Rolling off factor of the raised cosine interpolation filter (when PNCodeSampPerSym is x1)	0.5		Float	NO
RCFilterOrder	Number (even) of PN code symbols used in raised cosine filter	12		Integer	NO
CorrBlockSize	Number of oversampled samples in each sub correlation block	4		Integer	NO
PeakThreshold	Correlation coefficient threshold for peak search (0,1]	0.7		Float	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	Input	Input signal	complex	NO
2	PNInput	Local PN samples	complex	YES

Output Ports

Port	Name	Description	Signal Type	Optional
3	Corr	Correlation magnitude of input and local PN samples	real	NO
4	CarrierOffset	Normalized carrier offset ((Fc(Rx)-Fc(Tx)/Fsymbol)	real	NO
5	Output	Synchronized output signal aligned with integeral times of FrameSize (carrier offset not compensated)	complex	NO
6	OutputEnable	Synchronization validation flag of output signal	int	NO
7	SyncIndex	Number of input samples from real frame start to ideally aligned frame start	int	NO

Notes/Equations

1. This model completes the timing synchronization based on PN sequence correlation. It can detect the correct timing position and estimate the carrier offset. It can provide the correlation values, aligned output, output enable indicator, timing offset values and carrier offset.
2. x1, x2, ... x128 of OversampleRatio means the actual sampling ratio. Parameter FrameSize is a non-oversample value, the actual samples number of a frame still need to multiply $2^{\text{OversampleRatio}}$.
3. PN sequence can be inputted by PNInput pin or parameter PNCode. The PNInput pin is an optional pin, if it is connected, the PN sequency is inputted by it; otherwise, PN

sequence is inputted by parameter PNCode.

4. Multiple PN patterns can be supported by this model. For example, there are two PN patterns P1 and P2 with the same length N. They are used alternately in the heads of frames.

   Frame1: [$P1_1$, $P1_2$, ... ,$P1_N$, $Data(1)_1$, $Data(1)_2$, ... ,$Data(1)_{FrameSize-N}$]

   Frame2: [$P2_1$, $P2_2$, ... ,$P2_N$, $Data(2)_1$, $Data(2)_2$, ... ,$Data(2)_{FrameSize-N}$]

   Frame3: [$P1_1$, $P1_2$, ... ,$P1_N$, $Data(3)_1$, $Data(3)_2$, ... ,$Data(3)_{FrameSize-N}$]

   Frame4: [$P2_1$, $P2_2$, ... ,$P2_N$, $Data(4)_1$, $Data(4)_2$, ... ,$Data(4)_{FrameSize-N}$]

   ......

   where $P1_i$ is the i^{th} sample of P1, $P2_i$ is the i^{th} sample of P2, $Data(i)_j$ is the data in the j^{th} sample of i^{th} frame.

   Then the parameters for PN input can be specified as:

   PNCodePatternNum = 2

   PNCodeLength = N

   If PNInput pin is use, the first 2*N samples of PNInput pin should be [$P1_1$, $P1_2$, ... ,$P1_N$, $P2_1$, $P2_2$, ... ,$P2_N$].

   If parameter PNCode is used, PNCode = [$P1_1$, $P1_2$, ... ,$P1_N$, $P2_1$, $P2_2$, ... ,$P2_N$].

5. The oversampled PN sequence can also be used through specifying parameter PNCodeSampPerSym. In above example, if you want to use a oversampled sequence of P1 and P2, the oversample ratio can be specified as x1, x2, x4 and x8. For example, you have a four times oversampled sequence of P1 and P2 and you want to use it as the reference sequence for better synchronization performance. You can specify PNCodeSampPerSym as x4 and input the oversampled sequence from PNInput pin or parameter PNCode. But the size of input PN sequence should become from 2*N to 4*2*N.
6. When the PN sequence is non-oversampled (PNCodeSampPerSym = x1) and the inputted signal is also non-oversampled, the correlation of them may have an unconspicuous peak
   or even no peak, because the sampling position of the input signal is random within a non-oversampled period. So when the PN sequence is non-oversampled, it is interpolated to four times oversampled sequence automatically to improve the sampling resolution in this model.

   The interpolation is done according to Raised Cosine interpolation filter. The order of the Raised Cosine interpolation filter can be specified as an even in parameter RCFilterOrder and the roll-off factor of the Raised Cosine filter can be specified in parameter RCRolloffFactor.
7. The oversample ratios difference between input signal and PN sequence is supported.

   If the PN sequence oversample ratio is larger than that of input signal, for example, PN input sequence is non-oversampled and it is interpolated to four times oversampled sequence automatically, then there are four different samples within one non-oversample PN sampling period. The input signal is non-oversampled. Then each input signal sample, four correlation values will be calculated for four different oversampling positions. Output pin Corr will output 4 values for each input sample. It will search the correlation peak from all corrlation values and check which input sample index the correlation peak belongs to. Then it can find the timing synchronizaiton position.

   Otherwise, if the PN sequence oversample ratio is not larger than that of input signal, it calculates the correlation value once an input sample.
8. If PNInput pin is used, PNCodeSampPerSym can't be larger than OversampleRatio. Because when PNInput pin is used, it must start calculating the correlation values after the PN sequence input is finished. So if PNCodeSampPerSym is larger than OversampleRatio, it will miss the synchronization position of the first frame.
9. Because the input signal can be effected by carrier offset, it needs specify parameter CorrBlockSize to calculate the correlation values in many small blocks to relieve the impact of carrier offset. CorrBlockSize must be divided by PNLength exactly.

For instance, when PNCodeSampPerSym=OversampleRatio, PNLengthOS=PNLength*$2^{PNCodeSampPerSym}$. The PN sequence is [P_0, P_1, ... ,$P_{PNLengthOS-1}$] and input signal is [Input(n-PNLengthOS+1), Input(n-PNLengthOS+2), ... ,Input(n)]. Then the output of Corr output pin is

$$Corr(n) = \frac{\left|\sum_{j=0}^{BlockNum-2} X(j+1)\cdot conj(X(j))\right|}{\left|\sum_{j=0}^{BlockNum-2} Y(j+1)\cdot conj(Y(j))\right|}$$

where

$$X(j) = \sum_{i=0}^{CorrBlockSize-1} Input(i + j\cdot CorrBlockSize + n - PNLengthOS + 1)\cdot conj(P_{i+j\cdot CorrBlockSize})$$

$$Y(j) = \sum_{i=0}^{CorrBlockSize-1} P_{i+j\cdot CorrBlockSize}\cdot conj(P_{i+j\cdot CorrBlockSize})$$

$$BlockNum = \frac{PNLengthOS}{CorrBlockSize}, \qquad j = 0 \sim BlockNum - 1$$

So when CorrBlockSize is small, it can relieve the impact of carrier offset, but it needs more calculation.

10. Output pins are similar to those of CP Sync block. The peaks detection mode is similar to the Burst mode of CP sync model. It begins with an acquiring phase and searches the correlation peak. When it finds the correlation peak, it will stop the correlation calculation until the end of this burst. Then in the next frame, it searches the correlation peak again and get the new synchronization position.See *OFDM_CP_Sync* (algorithm) for details.

References

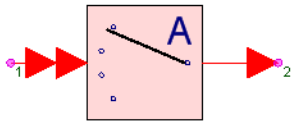
AsyncCommutator Part

Categories: *C++ Code Generation* (algorithm), *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AsyncCommutator (algorithm)	Asynchronous Data Commutator

AsyncCommutator (Asynchronous Data Commutator)



Description: Asynchronous Data Commutator
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *AsyncCommutator Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
BlockSizes	Block sizes read from each input	1		Integer array	NO	[1:∞)†

Input Ports

Port	Name	Signal Type	Optional
1	input	multiple anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim) or *variant* (sim).

Notes/Equations

1. AsyncCommutator takes N input streams, where N is the input bus width, and asynchronously combines them into one output stream.
2. B_i input samples are consumed from input #i (i = 1, ... , N), where B_i is an element of the BlockSizes parameter.
3. $B_1 + B_2 + \ldots + B_N$ samples are output. The first B_1 samples on the output come from the first input, the next B_2 samples come from the second input, and so on.

See:
Commutator (algorithm)
AsyncDistributor (algorithm)
Distributor (algorithm)

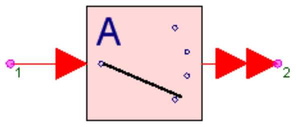
AsyncDistributor Part

Categories: *C++ Code Generation* (algorithm), *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AsyncDistributor (algorithm)	Asynchronous Data Distributor

AsyncDistributor (Asynchronous Data Distributor)



Description: Asynchronous Data Distributor
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *AsyncDistributor Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
BlockSizes	Block sizes written to each output	1		Integer array	NO	[1:∞)†

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	multiple anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim) or *variant* (sim).

Notes/Equations

1. AsyncDistributor asynchronously splits one input stream into N output streams. N, the output bus width, is the number elements in the BlockSizes parameter.
2. $B_1 + B_2 + \ldots + B_N$ samples are consumed from the input, where B_i (i = 1, ... , N) are the elements of BlockSizes.
3. B_i samples are distributed to output#i (i = 1, ... , N). The samples on the first output are the first B_1 samples of the input, the samples on the second output are the next B_2 samples of the input, and so on.

See:
Distributor (algorithm)
AsyncCommutator (algorithm)
Commutator (algorithm)

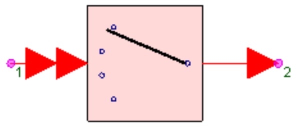
Commutator Part

Categories: *C++ Code Generation* (algorithm), *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Commutator (algorithm)	Synchronous Data Commutator

Commutator (Synchronous Data Commutator)



Description: Synchronous Data Commutator
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *Commutator Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
BlockSize	Number of particles in a block	1		Integer	NO	[1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	multiple anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim) or *variant* (sim).

Notes/Equations

1. This component takes N input streams and synchronously combines them into one output stream.
2. B samples are consumed from each input (where B is BlockSize), and N × B samples are output. The first B samples on the output come from the first input, the next B samples come from the second input, and so on.

See:
AsyncCommutator (algorithm)
Distributor (algorithm)
AsyncDistributor (algorithm)

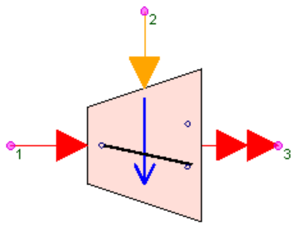
DeMux Part

Categories: *C++ Code Generation* (algorithm), *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
DeMux (algorithm)	Data Demultiplexer

DeMux (Data Demultiplexer)



Description: Data Demultiplexer
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *DeMux Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
BlockSize	Number of data items in a block	1		Integer	NO	[1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO
2	control	int	NO

Output Ports

Port	Name	Signal Type	Optional
3	output	multiple anytype	NO

Notes/Equations

1. Input should be demultiplexed to one of N output streams.
2. B samples are consumed from the input, where B is the BlockSize value, and copied to one output as determined by the control input. All other outputs get B zeroes of the appropriate type.
3. Integer values from 0 to N - 1 are expected at the control input. If the control input is outside this range, all outputs get B zeroes of the appropriate type.

See:
Mux (algorithm)

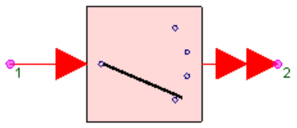
Distributor Part

Categories: *C++ Code Generation* (algorithm), *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Distributor (algorithm)	Synchronous Data Distributor

Distributor (Synchronous Data Distributor)



Description: Synchronous Data Distributor
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *Distributor Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
BlockSize	Number of data items in a block	1		Integer	NO	[1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	multiple anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim) or *variant* (sim).

Notes/Equations

1. Distributor synchronously splits one input stream into N output streams. N is equal to the number of connections to the output pin.
2. N × B samples (where B = BlockSize) are input, and distributes the first B samples to the first output, the next B samples to the next output, and so on.
3. Output has the same type as input.

See:
AsyncDistributor (algorithm)
Commutator (algorithm)
AsyncCommutator (algorithm)

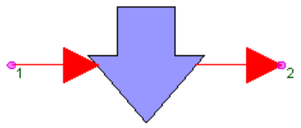
DownSample Part

Categories: *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
DownSample (algorithm)	Downsampler
DownSampleEnv (algorithm)	Down Sampler for Envelope Signal
DownSampleFxp (hardware)	Fixed Point DownSample

DownSample (Downsampler)



Description: Downsampler
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *DownSample Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Factor	Downsample factor	2		Integer	NO	[1:∞)
Phase	Downsample phase	0		Integer	NO	[0:Factor-1]

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	anytype	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	anytype	NO

⚠ To avoid confusion on a schematic, it is best to **mirror** (flip) this symbol (instead of **rotating** it) when you need to have the input on the *right* and the output on the *left* of the symbol; otherwise the symbol arrow will point in the "wrong" direction. Select the part and press F6 to flip the symbol into the correct orientation.

Notes/Equations

1. The DownSample model reduces the sampling rate of its input signal by an integer *Factor* ratio. Down-sampling is also referred to as *decimation*.
2. For every *Factor* samples received at the input, one sample is output.
3. This model does not have a built-in lowpass filter. Therefore, to avoid aliasing, it may be necessary to connect a lowpass filter at the input to ensure that the input signal bandwidth is appropriately limited.
4. The Phase parameter specifies which one sample (out of the Factor input samples read) to output: if Phase = 0, the first input sample is output; if Phase = Factor - 1, the latest input sample is output. The equation describing the behavior of this model is
   $y[n] = x[\text{Factor} \times n + \text{Phase}]$, where n is the output sample number, y is the output, and x is the input.

See:
DownSampleEnv (algorithm)
DownSampleVarPhase (algorithm)
UpSample (algorithm)

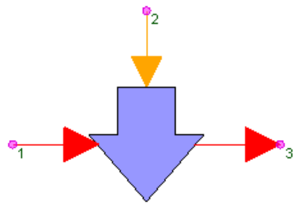
DownSampleVarPhase Part

Categories: *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
DownSampleVarPhase (algorithm)	Downsample with variable down sampling phase

DownSampleVarPhase (Downsample with Variable Downsampling Phase)



Description: Downsample with variable down sampling phase
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *DownSampleVarPhase Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Factor	Downsample factor	2		Integer	NO	[1:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	anytype	NO
2	phase	down sampling phase	int	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output signal	anytype	NO

⚠ To avoid confusion on a schematic, it is best to **mirror** (flip) this symbol (instead of **rotating** it) when you need to have the input on the *right* and the output on the *left* of the symbol; otherwise the symbol arrow will point in the "wrong" direction. Select the part and press F6 to flip the symbol into the correct orientation.

Notes/Equations

1. The DownSampleVarPhase model reduces the sampling rate of its input signal by an integer *Factor* ratio. Down-sampling is also referred to as *decimation*.
2. For every *Factor* samples received at the input, one sample is output.
3. This model does not have a built-in lowpass filter. Therefore, to avoid aliasing, it may be necessary to connect a lowpass filter at the input to ensure that the input signal bandwidth is appropriately limited.
4. The value of the phase input specifies which one sample (out of the Factor input samples read) to output: if phase = 0, the first input sample is output; if phase = Factor − 1, the latest input sample is output. The equation describing the behavior of this model is $y[n] = x[\text{Factor} \times n + \text{phase}]$, where n is the output sample number, y is the output, x is the value of the input input, and phase is the value of the phase input. If the value of the phase input is outside the range [0, Factor − 1], it is reset to 0 or Factor − 1 accordingly.

See:
DownSample (algorithm)
DownSampleEnv (algorithm)
UpSample (algorithm)

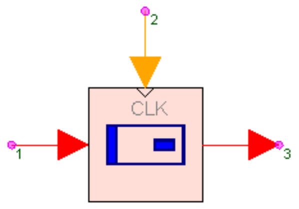
Latch Part

Categories: *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
Latch (algorithm)	Data Latch with Clock Control

Latch



Description: Data Latch with Clock Control
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Latch Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO
2	clock	int	NO

Output Ports

Port	Name	Signal Type	Optional
3	output	anytype	NO

Notes/Equations

1. The Latch model sets its *output* sample to the previously recorded *input* when the *clock* input was non-zero.
2. This model reads 1 sample from the *input* and *clock* and writes 1 sample to the *output*.
3. When the *clock* is non-zero, the previously recorded *input* when the *clock* input was non-zero is written to the *output* and the current *input* is saves for use the next time the *clock* is non-zero.

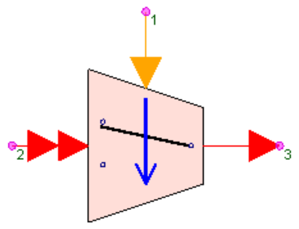
Mux Part

Categories: *C++ Code Generation* (algorithm), *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Mux (algorithm)	Data Multiplexer

Mux (Data Multiplexer)



Description: Data Multiplexer
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *Mux Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
BlockSize	Number of data items in a block	1		Integer	NO	[1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	control	int	NO
2	input	multiple anytype	NO

Output Ports

Port	Name	Signal Type	Optional
3	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim) or *variant* (sim).

Notes/Equations

1. One of N inputs is multiplexed to the output. N is the number of input connections.
2. BlockSize samples are consumed on each input. Only one of these blocks as determined by the control input is copied to the output.
3. Integer values from 0 through N - 1 are expected at the control input. If the control input is outside this range, the simulation is terminated with an error message.

See:
DeMux (algorithm)

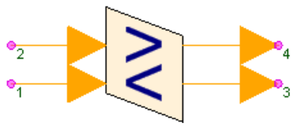
OrderTwoInt Part

Categories: *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
OrderTwoInt (algorithm)	Ordered Two Integer Min/Max Function

OrderTwoInt (Ordered Two Integer Min/Max Function)



Description: Ordered Two Integer Min/Max Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *OrderTwoInt Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	upper	int	NO
2	lower	int	NO

Output Ports

Port	Name	Signal Type	Optional
3	greater	int	NO
4	lesser	int	NO

Notes/Equations

1. OrderTwoInt takes two inputs and outputs the greater and lesser of the two inputs.
   $y_1 = \max(x_1, x_2)$
   $y_2 = \min(x_1, x_2)$
   where:
   x_1 is the upper input
   x_2 is the lower input
   y_1 is the greater output
   y_2 is the lesser output
2. For an input sample, one sample is output.

See:
Mux (algorithm)
DeMux (algorithm)

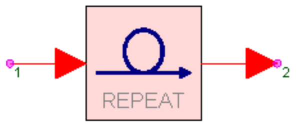
Repeat Part

Categories: *C++ Code Generation* (algorithm), *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Repeat (algorithm)	Data Repeater

Repeat



Description: Data Repeater
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *Repeat Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NumTimes	Repetition factor	2		Integer	NO	[1:∞)
BlockSize	Number of data items in a block	1		Integer	NO	[1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim) or *variant* (sim).

Notes/Equations

1. Repeat copies BlockSize number of inputs to the output NumTimes.
2. For every BlockSize number of inputs, there are BlockSize × NumTimes number of outputs.
3. This model is useful when a block of inputs needs upsampling.

Note on large upsampling factors

Each repeater require a buffer of NumTimes number of BlockSize × samples. For a large NumTimes value, memory requirements may prevent a simulation. The way around this problem is to substitute a cascade of repeaters. For example, a NumTimes of 10^{6} would require a buffer of 10^{6} BlockSize × samples. If a cascade of two repeater were used, then each NumTimes could be 10^{3} which would require a total buffer equivalent of 2×10^{3} BlockSize × samples. If a cascade of three repeater were used, then each NumTimes could be 10^{2} which would require a total buffer equivalent of 3×10^{2} BlockSize × samples.

See:
DownSample (algorithm)
UpSample (algorithm)

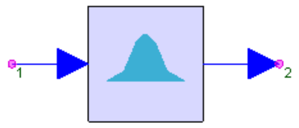
ResamplerRC Part

Categories: *C++ Code Generation* (algorithm), *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
ResamplerRC (algorithm)	Resampler with Raised Cosine Filter

ResamplerRC (Resampler with Raised Cosine Filter)



Description: Resampler with Raised Cosine Filter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *ResamplerRC Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Decimation	Decimation ratio	1		Integer	NO
DecimationPhase	Decimation phase	0		Integer	NO
Interpolation	Interpolation ratio	16		Integer	NO
Length	Number of taps	64		Integer	NO
ExcessBW	Excess bandwidth	1		Float	YES

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. The ResamplerRC implements a rational ratio resampler that uses a raised cosine filter as the interpolating filter.
2. For every Decimation number of input samples, Interpolation number of filtered values are output.
3. For more information on the multi-rate implementation, see *FIR* (algorithm).
4. DecimationPhase is equivalent to the Phase parameter of the *DownSample* (algorithm) part. If DecimationPhase is 0 (default), the earliest outputs of the decimation block are decimated.
5. The Length parameter defines the number of taps in the raised cosine filter.
6. ExcessBW parameter defines the excess bandwidth for the raised cosine filter. This value is often referred as the rolloff or the alpha of the raised cosine filter.
7. The impulse response of an ideal raised cosine filter is given by

$$mean[n] = \frac{1}{(n+1)\cdot BlockSize} \cdot \sum_{i=o}^{(n+1)\cdot BlockSize-1} in[i]$$

$$variance[n] = \frac{1}{(n+1)\cdot BlockSize} \cdot \sum_{i=o}^{(n+1)\cdot BlockSize-1} in^2[i] - mean^2[n]$$

The ideal impulse response is centered at 0. Since only causal filters are implemented, the actual impulse response used is

$$g[n] = h[n-M], \text{where } M = \begin{cases} \frac{L}{2}, & \text{if } L \text{ is even} \\ \frac{L-1}{2}, & \text{if } L \text{ is odd} \end{cases}$$

The impulse response is truncated outside the range [0, L-1].

References

1. E. A. Lee and D. G. Messerchmitt, *Digital Communication*, Kluwer Academic Publishers, Boston, 1988.
2. I. Korn, *Digital Communications*, Van Nostrand Reinhold, New York, 1985.

See:
FIR (algorithm)

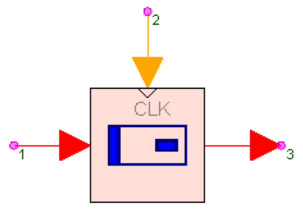
SampleHold Part

Categories: *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
SampleHold (algorithm)	Sample and hold with clock control

SampleHold



Description: Sample and hold with clock control
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *SampleHold Part* (algorithm)

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO
2	clock	int	NO

Output Ports

Port	Name	Signal Type	Optional
3	output	anytype	NO

Notes/Equations

1. The SampleHold model sets its *output* to the *input* when the *clock* is non-zero.
2. This model reads 1 sample from the *input* and *clock* and writes 1 sample to the *output*.

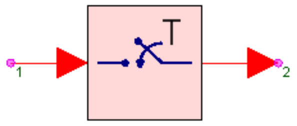
SetSampleRate Part

Categories: *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SetSampleRate (algorithm)	Set Signal Sample Rate

SetSampleRate (Set Signal Sample Rate)



Description: Set Signal Sample Rate
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *SetSampleRate Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
SampleRate	Sample rate	Sample_Rate	Hz	Float	NO	(0:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Notes/Equations

1. The SetSampleRate model associates a sample rate with the input data and sets the output with that data and its associated sample rate.
2. This model reads 1 sample from the input and writes 1 sample to the output.
3. The output sample rate generates time values for the output.
4. For a discussion on the precision of the time values, see *Sources Category* (algorithm).
5. A consistency check on the multirate graph will flag implementation problems.

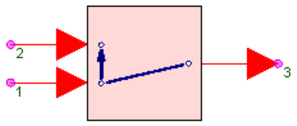
Trainer Part

Categories: *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Trainer (algorithm)	Initial Sample Trainer

Trainer (Initial Sample Trainer)



Description: Initial Sample Trainer
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Trainer Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
TrainLength	Number of training samples to use	100		Integer	NO	[0:∞)

Input Ports

Port	Name	Signal Type	Optional
1	train	anytype	NO
2	decision	anytype	NO

Output Ports

Port	Name	Signal Type	Optional
3	output	anytype	NO

Notes/Equations

1. Trainer passes the value of the training input to the output for the first TrainLength samples, then passes the decision input to the output.
2. For every input, there is an output.
3. During the startup phase, the decision inputs are discarded. After the startup phase, the training inputs are discarded.
4. This component is designed for use with adaptive equalizers that require a training sequence at startup, but it can be used whenever one sequence is used during a startup phase, and another sequence after that.

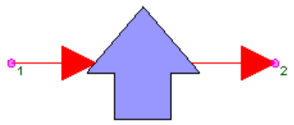
UpSample Part

Categories: *Routers/Resamplers* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
UpSample (algorithm)	Up Sampler
UpSampleEnv (algorithm)	Up Sampler for Envelope Signal
UpSampleFxp (hardware)	Fixed Point UpSample

UpSample (Upsampler)



Description: Up Sampler
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *UpSample Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Factor	Number of samples produced	2		Integer	NO	[1:∞)
Mode	Mode method: Insert zeros, Hold sample	Insert zeros		Enumeration	YES	
Phase	Where to put the input in the output block. Visible when Mode mode is Insert zeros	0		Integer	YES	[0:Factor-1]

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	anytype	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	anytype	NO

⚠ To avoid confusion on a schematic, it is best to **mirror** (flip) this symbol (instead of **rotating** it) when you need to have the input on the *right* and the output on the *left* of the symbol; otherwise the symbol arrow will point in the "wrong" direction. Select the part and press F6 to flip the symbol into the correct orientation.

Notes/Equations

1. UpSample increases the sample rate of the input signal by *Factor*.
2. This model reads one sample from the input and writes *Factor* number of samples to the output.
3. If Mode is *Insert zeros*, the inserted output samples have the zero value of the input type. Otherwise, the output samples have the same value as the most recent input.
4. For *Insert zeros* mode only, the Phase parameter specifies where to place the input sample in a block of Factor output samples. Phase number of zeros are output first followed by the input value followed by (Factor - 1) - Phase number of zeros. Examples follow.
    1. If Phase is zero, the input value is output first followed by Factor - 1 number of zeros.
    2. If Phase is Factor - 1, Factor - 1 number of zeros are output first followed by the input value.

Note on large upsampling factors

Each upsampler require a buffer of Factor number of samples. For a large Factor value, memory requirements may prevent a simulation. The way around this problem is to substitute a cascade of upsamplers. For example, a Factor of 10^6 would require a buffer of 10^6 samples. If a cascade of two upsamplers were used, then each Factor could be 10^3 which would require a total buffer equivalent of 2×10^3 samples. If a cascade of three upsamplers were used, then each Factor could be 10^2 which would require a total buffer equivalent of 3×10^2 samples.
If a nonzero Phase were to be used, the Phase would have to be deconstructed into Phase values for each cascaded upsampler. Let an upsample part have a Factor of 10^6 and a Phase of 123456. The upsampler is substituted by 3 upsamplers each having a Factor of 10^2. The first upsampler would have a Phase of 12, the second upsampler would have a Phase of 34 and the third sampler would have a Phase of 56.

See:
DownSample (algorithm)
Repeat (algorithm)

Signal Processing

The Signal Processing library provides parts that implement signal processing algorithms, like FFT, Quantization, Compress/Expand, etc.

Contents

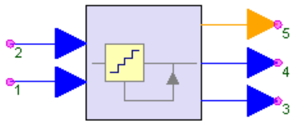
AdaptLinQuant Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AdaptLinQuant (algorithm)	Adaptive Linear Quantizer

AdaptLinQuant (Adaptive Linear Quantizer)



Description: Adaptive Linear Quantizer
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AdaptLinQuant Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Bits	Number of bits	8		Integer	YES	[1:31]

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO
2	inStep	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	amplitude	real	NO
4	outStep	real	NO
5	stepLevel	int	NO

Notes/Equations

1. Input is quantized to 2^{Bits} number of levels in a plan described as odd symmetric about zero midrise.
2. For each model execution, one sample is read at each input and one sample is written to each output.
3. Quantization levels are uniformly spaced by inStep and symmetric about zero. Therefore, the *high* quantization level is $(2^{\text{Bits}} - 1)(\text{inStep} / 2)$, the *low* quantization level is set to −*high*. Zero is not a quantization level.
4. Input is held to a value between *low* and *high* and rounded to the nearest quantization level.
5. Quantized input is output on the amplitude port. Its quantization index is output on the stepLevel port as an integer between 0 and $2^{\text{Bits} - 1}$.
6. inStep is copied to the outStep port.

See:
LinearQuantizer (algorithm)
Quantizer (algorithm)
Quantizer2D (algorithm)

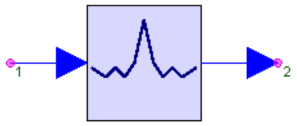
AutoCorr Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AutoCorr (algorithm)	Autocorrelation Estimator

AutoCorr (Autocorrelation Estimator)



Description: Autocorrelation Estimator
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AutoCorr Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
CorrelationType	Correlation method: NonCircular, Circular	Circular		Enumeration	YES		
CorrelationLength	Number of input samples	511		Integer	NO	[1:∞)	N
StartLag	Low lag limit to output	-255		Integer	NO	(-N:L<sub>h</sub>]	L_l
StopLag	High lag limit to output	255		Integer	NO	[L<sub>l</sub>:N)	L_h
Normalization	Correlation estimate normalization: None, UnBiased, Biased	None		Enumeration	YES		

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	real	NO

Notes/Equations

1. AutoCorr estimates the autocorrelation function of the input.
2. The part inputs N samples and outputs $L_l + L_h + 1$ autocorrelation values.
3. $\mathbf{r}_{\mathbf{x}}(k)$ is the estimated autocorrelation function of N input samples and is evaluated for a k from $L_l, \dots, L_h$.
4. $\mathbf{r}_{\mathbf{x}}(L_l)$ is output first, and $\mathbf{r}_{\mathbf{x}}(L_h)$ is output last.
5. If CorrelationType is *NonCircular*, then
   $\mathbf{r}_{\mathbf{x}}(k)$ = **sum** { **x** (i) **x** (i + k) } for input **x** and i = 1, ... , N. For p < 1 or p > N, **x** (p) is 0.
   Otherwise, the CorrelationType is *Circular*, then
   $\mathbf{r}_{\mathbf{x}}(k)$ = **sum** { **x** (i) **x** (i + k) } for input **x** and i = 1, ... , N. For p < 1 or p > N, **x** (p) is **x** (((p - 1) **modulo** N) + 1). Define -1 **modulo** N as N - 1, and so on.
6. Additionally, *UnBiased* and *Biased* autocorrelation estimates are supported.
   - If CorrelationType is *NonCircular*, then *for UnBiased* Normalization, $\mathbf{r}_{\mathbf{x}}(k)$ is divided by N - | k |. For *Biased* Normalization, $\mathbf{r}_{\mathbf{x}}(k)$ is divided by N.
   - If CorrelationType is *Circular*, then for *UnBiased* or *Biased* Normalization, $\mathbf{r}_{\mathbf{x}}(k)$ is divided by N.
7. A typical application of AutoCorr is to find periodicities in N samples of input. For this application, set CorrelationType to *Circular*.

See:
CrossCorr (algorithm)

References

1. A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall: Englewood Cliffs, NJ, 1989.

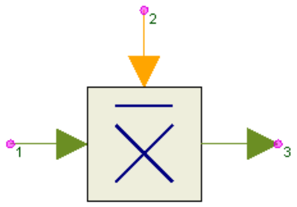
AverageCxWOffset Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
AverageCxWOffset (algorithm)	Complex Averager with Offset Control

AverageCxWOffset (Complex Averager with Offset Control)



Description: Complex Averager with Offset Control
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *AverageCxWOffset Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
NumInputsToAverage	Number of input samples to average	256		Integer	NO	[1:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	Input	Input	complex	NO
2	Offset	Offset	int	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	Output	Output	complex	NO

Notes/Equations

1. AverageCxWOffset averages a window of NumInputsToAverage samples.
2. For each model execution, one sample is input at Input and Offset, and one sample is output.
3. If NumInputsToAverage samples are not available, the last output is held. The initial hold value is zero.
4. Otherwise, the averaged value is output, and all samples in the window and older are considered stale and discarded.
5. The window of NumInputsToAverage samples is offset from the most recent inputs by the Offset value. The offset must be non-negative and less than NumSymToAverage.
6. AverageCxWOffset can be used to average RF received data using channel delay information. The output is the averaged complex signal envelope.

References

1. M. Jeruchim, P. Balaban and K. Shanmugan, "Simulation of Communication System," Plenum Press, New York and London, 1992.

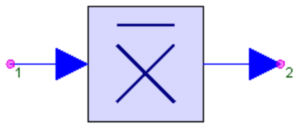
Average Part

Categories: *C++ Code Generation* (algorithm), *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Average (algorithm)	Averager
AverageCx (algorithm)	Complex Averager

Average (Averager)



Description: Averager
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Average Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
NumInputsToAverage	Number of input blocks to average	8		Integer	NO
BlockSize	Sample size of input blocks that are averaged to produce an output block	1		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. The Average model calculates the average over *NumInputsToAverage* real-valued input samples. If *BlockSize* is 1, then *NumInputsToAverage* samples are read from the input, averaged, and the average is written to the output. If *BlockSize* is greater than 1, then *NumInputsToAverage* × *BlockSize* samples are read from the input, *BlockSize* averages are computed by averaging the *NumInputsToAverage* samples at the same position (first, second, third, ..., B-th) in every block, and these *BlockSize* averages are written to the output. The output values are calculated based on the following equation:

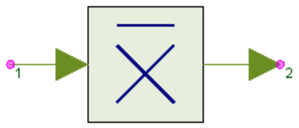
$$mean[n] = \frac{1}{(n+1)\cdot BlockSize}\cdot\sum_{i=0}^{(n+1)\cdot BlockSize-1} in[i]$$

$$variance[n] = \frac{1}{(n+1)\cdot BlockSize}\cdot\sum_{i=0}^{(n+1)\cdot BlockSize-1} in^2[i] - mean^2[n]$$

2. At every execution of this model, *NumInputsToAverage* × *BlockSize* samples are read from the input and *BlockSize* samples are written to the output.

See:
AverageCx (algorithm)

AverageCx (Complex Averager)



Description: Complex Averager
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Average Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
NumInputsToAverage	Number of input blocks to average	8		Integer	NO
BlockSize	Sample size of input blocks that are averaged to produce an output block	1		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	complex	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	complex	NO

Notes/Equations

1. The AverageCx model calculates the average over *NumInputsToAverage* complex-valued input samples. If *BlockSize* is 1, then *NumInputsToAverage* samples are read from the input, averaged, and the average is written to the output. If *BlockSize* is greater than 1, then *NumInputsToAverage* × *BlockSize* samples are read from the input, *BlockSize* averages are computed by averaging the *NumInputsToAverage* samples at the same position (first, second, third, ..., B-th) in every block, and these *BlockSize* averages are written to the output. The output values are calculated based on the following equation:

$$mean[n] = \frac{1}{(n+1)\cdot BlockSize} \cdot \sum_{i=0}^{(n+1)\cdot BlockSize-1} in[i]$$

$$variance[n] = \frac{1}{(n+1)\cdot BlockSize} \cdot \sum_{i=0}^{(n+1)\cdot BlockSize-1} in^2[i] - mean^2[n]$$

2. At every execution of this model, *NumInputsToAverage* × *BlockSize* samples are read from the input and *BlockSize* samples are written to the output.

See:
Average (algorithm)

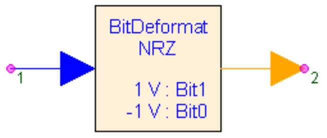
BitDeformatter Part

Categories: *C++ Code Generation* (algorithm), *Signal Processing* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
BitDeformatter (algorithm)	NRZ/RZ Symbol to Bit Converter

BitDeformatter (NRZ/RZ Symbol to Bit Converter)



Description: NRZ/RZ Symbol to Bit Converter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *BitDeformatter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
SamplesPerBit	Number of input samples per bit	1		Integer	NO
Format	Format for input signal: NRZ, RZ	NRZ		Enumeration	NO
LogicZeroLevel	Voltage for bit value zero	-1	V	Float	YES
LogicOneLevel	Voltage for bit value one	1	V	Float	YES

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	boolean	NO

Notes/Equations

1. The BitDeformatter converts input NRZ/RZ symbols to output bits.
2. This model reads SamplesPerBit samples from input representing one NRZ/RZ symbol and interprets it using the Format parameter to write 1 sample to output as a bit value that has a logic level defined by LogicZeroLevel and LogicOneLevel.
3. Format specifies the pulse-code modulation (PCM) waveform type: Nonreturn-to-zero (*NRZ*) or Return-to-zero (*RZ*). Unipolar RZ and Bipolar RZ are specified with appropriate values for parameters LogicZeroLevel and LogicOneLevel.
4. For *NRZ*, the sampled input waveform is averaged. The averaged value that is closer to LogicZeroLevel than LogicOneLevel outputs a zero bit, otherwise a one bit is output.
5. For *RZ*, the SamplerPerBit parameter must be a multiple of two as the format returns to "zero" in the last half of the waveform. The sampled input waveform is separately averaged in the first and second half of the bit period. The averaged "zero" provide a zero reference for the bit period. If the differential value for the bit period is closer to LogicZeroLevel than LogicOneLevel, a zero bit is output, otherwise a one bit is output.

See:
BitFormatter (algorithm)

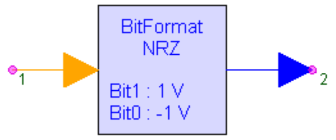
BitFormatter Part

Categories: *C++ Code Generation* (algorithm), *Signal Processing* (algorithm), *Type Converters* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
BitFormatter (algorithm)	Bit to NRZ/RZ Symbol Converter

BitFormatter (Bit to NRZ/RZ Symbol Converter)



Description: Bit to NRZ/RZ Symbol Converter
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *BitFormatter Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
SamplesPerBit	Number of output samples per input bit	1		Integer	NO
Format	Format for output signal: NRZ, RZ	NRZ		Enumeration	NO
LogicZeroLevel	Voltage for bit value zero	-1	V	Float	YES
LogicOneLevel	Voltage for bit value one	1	V	Float	YES

Input Ports

Port	Name	Signal Type	Optional
1	input	boolean	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. The BitFormatter converts input bits to output NRZ/RZ symbols.
2. This model reads 1 sample from the input representing one bit and interprets it using the Format parameter to write SamplesPerBit samples representing one NRZ/RZ symbol to the output with a level defined by LogicZeroLevel and LogicOneLevel.
3. Any input greater than zero maps to a one bit value, otherwise a zero bit is input.
4. For each bit, a pulse-code modulation (PCM) waveform is generated. The SamplesPerBit parameter determines the waveform period.
5. The Format parameter determines the PCM type: Nonreturn-to-zero (*NRZ*) or Return-to-zero (*RZ*). Unipolar RZ and Bipolar RZ are represented by selecting appropriate values for LogicOneLevel and LogicZeroLevel parameters.
6. For *NRZ*, a one bit value input outputs a constant waveform at LogicOneLevel while a zero bit value input outputs a waveform at LogicZeroLevel.
7. For *RZ*, the SamplesPerBit parameter must be a multiple of two since the last half of the waveform must return to zero. For the first half of the waveform, a one bit value input outputs LogicOneLevel samples while a zero bit outputs LogicZeroLevel samples.

See:
BitDeformatter (algorithm)

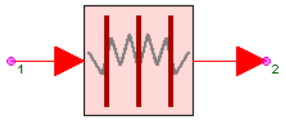
Chop Part

Categories: *C++ Code Generation* (algorithm), *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Chop (algorithm)	Data Block Chopper

Chop (Data Block Chopper)



Description: Data Block Chopper
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *Chop Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
nRead	Number of data items read	128		Integer	NO	[1:∞)
nWrite	Number of data items written	64		Integer	NO	[1:∞)
Offset	Start of output block relative to start of input block	0		Integer	NO	(-∞:∞)
UsePastInputs	Use previously read inputs: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim) or *variant* (sim).

Notes/Equations

1. The Chop model reads a block of *nRead* samples from its input and produces a block of *nWrite* samples at its output. The output block may have samples from the current or previous input block, may discard samples from the current block, and may append/prepend zeros to the input block.
2. The *Offset* parameter defines where in the output block of samples the first (oldest) input sample is output.
   - If *Offset* is ≤ 0, the first |*Offset*| input samples are discarded and the (|*Offset*| + 1)-th input sample is output as the first output sample (the *UsePastInputs* parameter is ignored)
   - If *Offset* > 0, the first input sample is output as the (*Offset* + 1)-th output sample. The first *Offset* output samples are:
     - 0, if *UsePastInputs* is set to NO
     - the last *Offset* samples from the previous blocks read, if *UsePastInputs* is set to YES
3. The following tables summarize the behavior of this component.
   If *nRead* ≥ *nWrite*

Case	***Offset***	***UsePastInputs***	***nWrite* ≤ *nRead* − |*Offset*|**	***Output***
1	(−∞, −*nRead*]	NO or YES	will always be FALSE	all zeros
2	(−*nRead*, 0]	NO or YES	TRUE	discard the first |*Offset*| input samples, output the next *nWrite* input samples
3	(−*nRead*, 0]	NO or YES	FALSE	discard the first |*Offset*| input samples, output the next (*nRead* − |*Offset*|) input samples followed by (*nWrite* − (*nRead* − |*Offset*|)) zeros
4	(0, *nWrite*)	NO	TRUE or FALSE	output *Offset* zeros followed by the first (*nWrite* − *Offset*) input samples
5	(0, *nWrite*)	YES	TRUE or FALSE	output the last *Offset* samples of the previously read input block followed by the first (*nWrite* − *Offset*) input samples
6	[*nWrite*, ∞)	NO	TRUE or FALSE	all zeros
7	[*nWrite*, ∞)	YES	TRUE or FALSE	output from the (*nRead* − *Offset* + 1)-th to (*nRead* − *Offset* + *nWrite*)-th samples of the previously read input block (for the first block the previous block is assumed to be all zeros)

If *nRead* < *nWrite*

Case	***Offset***	***UsePastInputs***	***nRead* ≤ *nWrite* − |*Offset*|**	***Output***
8	(−∞, −*nRead*]	NO or YES	TRUE or FALSE	all zeros
9	(−*nRead* , 0]	NO or YES	TRUE or FALSE	discard the first |*Offset*| input samples, output the next (*nRead* − |*Offset*|) input samples followed by (*nWrite* − (*nRead* − |*Offset*|)) zeros
10	(0, *nWrite*)	NO	TRUE	output *Offset* zeros followed by the *nRead* input samples followed by (*nWrite* − *nRead* − *Offset*) zeros
11	(0, *nWrite*)	NO	FALSE	output *Offset* zeros followed by the first (*nWrite* − *Offset*) input samples
12	(0, *nWrite*)	YES	TRUE	output the last *Offset* samples of the previously read input block followed by the *nRead* input samples followed by (*nWrite* − *nRead* − *Offset*) zeros
13	(0, *nWrite*)	YES	FALSE	output the last *Offset* samples of the previously read input block(s) followed by the first (*nWrite* − *Offset*) input samples
14	[*nWrite*, ∞)	NO	will always be FALSE	all zeros
15	[*nWrite*, ∞)	YES	will always be FALSE	output the last *nWrite* samples of the previously read input block(s) (for the first block the previous blocks are assumed to be all zeros)

4. Here are some examples. In all of these examples the input is assumed to be a ramp signal with initial value of 1 and step 1 (1, 2, 3, 4, 5, 6, ...).

Case	***nRead***	***nWrite***	***Offset***	***UsePastInputs***	***Output***
1	10	5	-10 (or smaller)	NO or YES	0, 0, 0, 0, 0, 0, ...
2	10	3	-5	NO or YES	6, 7, 8, 16, 17, 18, 26, 27, 28, ...
3	10	5	-7	NO or YES	8, 9, 10, 0, 0, 18, 19, 20, 0, 0, 28, 29, 30, 0, 0, ...
4	10	5	2	NO	0, 0, 1, 2, 3, 0, 0, 11, 12, 13, 0, 0, 21, 22, 23, ...
5	10	5	2	YES	0, 0, 1, 2, 3, 9, 10, 11, 12, 13, 19, 20, 21, 22, 23, ...
6	10	5	5 (or bigger)	NO	0, 0, 0, 0, 0, 0, ...
7	10	5	5	YES	0, 0, 0, 0, 0, 6, 7, 8, 9, 10, 16, 16, 18, 19, 20, ...
7	10	5	7	YES	0, 0, 0, 0, 0, 4, 5, 6, 7, 8, 14, 15, 16, 17, 18, ...
8	5	10	-5 (or smaller)	NO or YES	0, 0, 0, 0, 0, 0, ...
9	5	10	-3	NO or YES	4, 5, 0, 0, 0, 0, 0, 0, 0, 0, 9, 10, 0, 0, 0, 0, 0, 0, 0, 0, 14, 15, 0, 0, 0, 0, 0, 0, 0, 0, ...
10	5	10	3	N0	0, 0, 0, 1, 2, 3, 4, 5, 0, 0, 0, 0, 0, 6, 7, 8, 9, 10, 0, 0, 0, 0, 0, 11, 12, 13, 14, 15, 0, 0, ...
11	5	10	7	N0	0, 0, 0, 0, 0, 0, 0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 6, 7, 8, 0, 0, 0, 0, 0, 0, 0, 11, 12, 13, ...
12	5	10	3	YES	0, 0, 0, 1, 2, 3, 4, 5, 0, 0, 3, 4, 5, 6, 7, 8, 9, 10, 0, 0, 8, 9, 10, 11, 12, 13, 14, 15, 0, 0, ...
13	5	10	7	YES	0, 0, 0, 0, 0, 0, 0, 1, 2, 3, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ...
14	5	10	10 (or bigger)	NO	0, 0, 0, 0, 0, 0, ...
15	3	5	5	YES	0, 0, 0, 0, 0, 0, 0, 1, 2, 3, 2, 3, 4, 5, 6, 5, 6, 7, 8, 9, ...

5. Common uses of the Chop component include:

- Discard samples from the beginning of a block of data: *Offset* should be set to $-N$, where N is the number of samples that need to be discarded and *nWrite* should be set to *nRead* $- N$.
- Discard samples from the end of a block of data: *Offset* should be set to 0 and *nWrite* should be set to *nRead* $- N$, where N is the number of samples that need to be discarded.
- Discard samples from both the beginning and the end of a block of data: *Offset* should be set to $-N_b$, where N_b is the number of samples that need to be discarded from the beginning of the block and *nWrite* should be set to *nRead* $- N_b - N_e$, where N_e is the number of samples that need to be discarded from the end of the block.
- Prepend zeros to a block of data: *Offset* should be set to N, where N is the number of zeros to be prepended, *UsePastInputs* should be set to NO, and *nWrite* should be set to *nRead* $+ N$.
- Append zeros to a block of data: *Offset* should be set to 0 and *nWrite* should be set to *nRead* $+ N$, where N is the number of zeros to be appended.
- Prepend and append zeros to a block of data: *Offset* should be set to N_p, where N_p is the number of zeros to be prepended, *UsePastInputs* should be set to NO, and *nWrite* should be set to *nRead* $+ N_p + N_a$, where N_a is the number of zeros to be appended.
- Break an input stream of samples in blocks of size N_b with N_o overlapping samples: *nRead* should be set to $N_b - N_o$, *nWrite* should be set to N_b, *Offset* should be set to N_o, and *UsePastInputs* should be set to YES.

See:
ChopVarOffset (algorithm)

Timing

When this model is used in a timed simulation, the first sample at its output will occur at time t_0 + *(nRead - 1) / SampleRate*, where t_0 is the time stamp of the first input sample and *SampleRate* is the sample rate of the input signal. For more information about timing see *Timing Method* (sim).

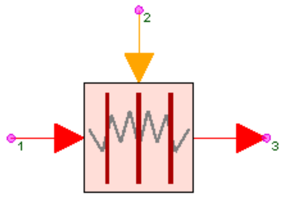
ChopVarOffset Part

Categories: *C++ Code Generation* (algorithm), *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
ChopVarOffset (algorithm)	Data Block Chopper with Offset Control

ChopVarOffset (Data Block Chopper with Offset Control)



Description: Data Block Chopper with Offset Control
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *ChopVarOffset Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
nRead	Number of data items read	128		Integer	NO	[1:∞)
nWrite	Number of data items written	64		Integer	NO	[1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO
2	offsetCntrl	int	NO

Output Ports

Port	Name	Signal Type	Optional
3	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim) or *variant* (sim).

Notes/Equations

1. ChopVarOffset has the same functionality as the *Chop* (algorithm) model except that the *Offset* parameter is determined at run time by a control input (*offsetCntrl*) and the *UsePastInputs* parameter is assumed to be NO.
2. The model reads a block of *nRead* samples from its input and produces a block of *nWrite* samples at its output.

See:
Chop (algorithm)

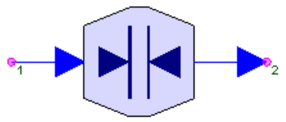
Compress Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Compress (algorithm)	Compression Part of a Compander

Compress (Compression Part of a Compander)



Description: Compression Part of a Compander
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Compress Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
CompressionType	Compression law: MU-law, A-law	MU-law		Enumeration	YES	
CompressionK	Compression constant	1		Float	YES	
Max	Maximum input value magnitude	1		Float	YES	(0.0:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

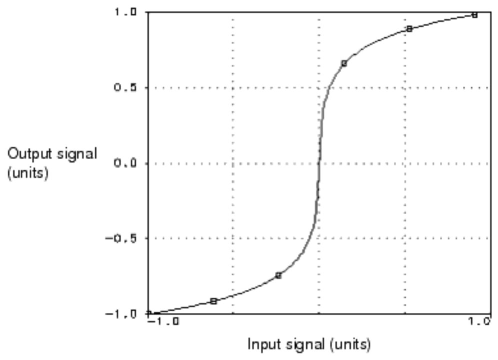
Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. Compress performs MU-law or A-law compression.
2. For every input, there one output.
3. Let x′(n) = x(n) / Max, then
   MU-law:
   $$y(n) = V_M \frac{\operatorname{sgn}[x'(n)] \ln\{1.0 + \mu|x'(n)|\}}{\ln(1.0+\mu)} \text{ for } \mu \ge 0$$
   A-law:
   $$y(n) = \begin{cases} V_M \dfrac{\operatorname{sgn}[x'(n)] A|x'(n)|}{1.0 + \ln(A)} & \text{for } |x'(n)| < 1/A \\ V_M \dfrac{\operatorname{sgn}[x'(n)]\{1 + \ln[A|x'(n)|]\}}{1 + \ln(A)} & \text{for } |x'(n)| \ge 1/A \end{cases}$$
   where
   x(n) is input for sample n
   V_M is Max
   y(n) is output for sample n
   μ is CompressionK for *MU-law*
   A is CompressionK for *A-law*
4. Shown below is the input/output characteristic of Compress with Type = *MU-law*, CompressionK = 255 and Max = 1V.

   Compress Signal Plot



See:
Expand (algorithm)

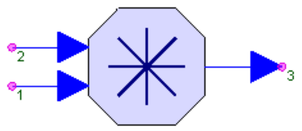
Convolve Part

Categories: *C++ Code Generation* (algorithm), *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Convolve (algorithm)	Convolution Function
ConvolveCx (algorithm)	Complex Convolution Function

Convolve (Convolution Function)



Description: Convolution Function
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Convolve Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
TruncationDepth	Maximum number of terms in convolution sum	256		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	inA	real	NO
2	inB	real	NO

Output Ports

Port	Name	Signal Type	Optional
3	out	real	NO

Notes/Equations

1. The Convolve model convolves two real-valued causal finite length sequences.

$$mean[n] = \frac{1}{(n+1)\cdot BlockSize} \cdot \sum_{i=0}^{(n+1)\cdot BlockSize-1} in[i]$$

$$variance[n] = \frac{1}{(n+1)\cdot BlockSize} \cdot \sum_{i=0}^{(n+1)\cdot BlockSize-1} in^2[i] - mean^2[n]$$

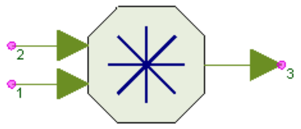
2. For each model execution, one sample is read from inA and inB, and one convolution value is output.
3. TruncationDepth must be set to a value larger than the number of output samples of interest. Otherwise, the results will be unexpected after TruncationDepth samples.
4. If one input has finite length and does not change over time, whereas the other input can be arbitrarily long, use the *FIR* (algorithm) model. Set the Taps parameter of the *FIR* (algorithm) model to the values of the finite length sequence.
5. If one input has finite length and changes over time, whereas the other input can be arbitrarily long, use the *BlockFIR* (algorithm) model. *BlockFIR* (algorithm) allows filtering of a signal in fixed size blocks, where each input block is filtered with a different set of coefficients.

See:
ConvolveCx (algorithm)

References

1. A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall: Englewood Cliffs, NJ, 1989.

ConvolveCx (Complex Convolution Function)



Description: Complex Convolution Function
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *Convolve Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
TruncationDepth	Maximum number of terms in convolution sum	256		Integer	NO

Input Ports

Port	Name	Signal Type	Optional
1	inA	complex	NO
2	inB	complex	NO

Output Ports

Port	Name	Signal Type	Optional
3	out	complex	NO

Notes/Equations

1. The ConvolveCx model convolves two complex-valued causal finite length sequences.

$$mean[n] = \frac{1}{(n+1)\cdot BlockSize} \cdot \sum_{i=0}^{(n+1)\cdot BlockSize-1} in[i]$$

$$variance[n] = \frac{1}{(n+1)\cdot BlockSize} \cdot \sum_{i=0}^{(n+1)\cdot BlockSize-1} in^2[i] - mean^2[n]$$

2. For each model execution, one sample is read from inA and inB, and one convolution value is output.
3. TruncationDepth must be set to a value larger than the number of output samples of interest. Otherwise, the results will be unexpected after TruncationDepth samples.
4. If one input has finite length and does not change over time, whereas the other input can be arbitrarily long, use the *FIR_Cx* (algorithm) model. Set the Taps parameter of the *FIR_Cx* (algorithm) model to the values of the finite length sequence.

See:
Convolve (algorithm)

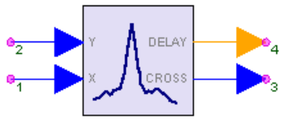
CrossCorr Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
CrossCorr (algorithm)	Cross Correlator

CrossCorr (Cross Correlator)



Description: Cross Correlator
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *CrossCorr Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
CorrelationType	Correlation method: NonCircular, Circular	NonCircular		Enumeration	YES		
CorrelationLength	Number of input samples	511		Integer	NO	[1:∞)	N
StartLag	Low lag limit to output	-255		Integer	NO	(-N:L<sub>h</sub>]	L_l
StopLag	High lag limit to output	255		Integer	NO	[L<sub>l</sub>:N)	L_h
Normalization	Correlation estimate normalization: None, UnBiased, Biased	None		Enumeration	YES		

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	real	NO
2	input2	second input signal	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	output	output signal	real	NO
4	delay	delay of input2 with respect to input	int	NO

Notes/Equations

1. CrossCorr estimates the cross correlation function for two inputs.
2. The part inputs N samples from pin 1 and from pin 2 and outputs $L_l + L_h + 1$ cross correlation values to pin 3 and one delay value to pin 4.
3. $\mathbf{r}_{\mathbf{xy}}(k)$ is the estimated cross correlation function of inputs from pin 1 (**x**) and pin 2 (**y**) and is evaluated for a k from L_l, ... , L_h.
4. $\mathbf{r}_{\mathbf{xy}}(L_l)$ is output first, and $\mathbf{r}_{\mathbf{xy}}(L_h)$ is output last.
5. If CorrelationType is *NonCircular*, then
$\mathbf{r}_{\mathbf{xy}}(k) = \mathbf{sum}\{\mathbf{x}(i)\,\mathbf{y}(i+k)\}$ for i = 1, ... , N. For p < 1 or p > N, **x** (p) and **y** (p) are 0.
Otherwise, the CorrelationType is *Circular*, then
$\mathbf{r}_{\mathbf{xy}}(k) = \mathbf{sum}\{\mathbf{x}(i)\,\mathbf{y}(i+k)\}$ for i = 1, ... , N. For p < 1 or p > N, **x** (p) is **x** (q) and **y** (p) is **y** (q) where q is ((p − 1) **modulo** N) + 1). Define −1 **modulo** N as N − 1, and so on.

6. Additionally, *UnBiased* and *Biased* autocorrelation estimates are supported.
   - If CorrelationType is *NonCircular*, then *for UnBiased* Normalization, **r** $_{\mathbf{xy}}$ (k) is divided by N − | k |. For *Biased* Normalization, **r** $_{\mathbf{xy}}$ (k) is divided by N.
   - If CorrelationType is *Circular*, then for *UnBiased* or *Biased* Normalization, **r** $_{\mathbf{xy}}$ (k) is divided by N.
7. The estimated delay in samples of **y** with respect to **x** is output at pin 4. A negative value imply that **x** is delayed with respect to **y**.
8. A typical application for CrossCorr is to find the best delay that will synchronize two signals. For this application set CorrelationType to *NonCircular*.

See:
AutoCorr (algorithm)

References

1. A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall: Englewood Cliffs, NJ, 1989.

DeadZone Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
DeadZone (algorithm)	Dead Zone Nonlinearity

DeadZone (Dead Zone Nonlinearity)

Description: Dead Zone Nonlinearity
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *DeadZone Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
K	Magnitude gain	1		Float	YES	(-∞:0.0) or (0.0:∞)
Low	Lower dead zone value	0		Float	YES	(-∞:High)
High	Higher dead zone value	1		Float	YES	(-∞:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	real	NO

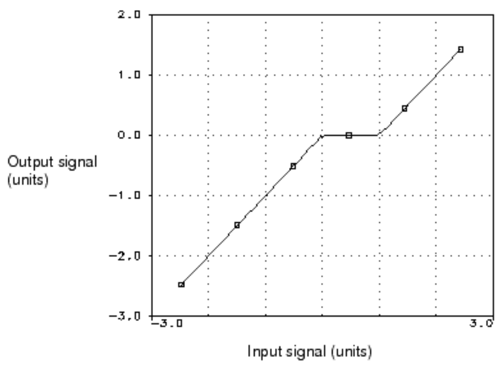
Notes/Equations

1. The dead zone nonlinearity is defined as follows:

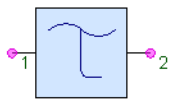
$$y(n) = \begin{cases} K(x(n) - V_h) & \text{for} \quad x(n) > V_h \\ K(x(n) - V_l) & \text{for} \quad x(n) < V_l \\ 0 & \text{otherwise} \end{cases}$$

   where:
   x(n) is input for sample n
   K is the K parameter
   V_h is the High parameter
   V_l is the Low parameter
   y(n) is output for sample n
2. For every input, there is one output.
3. Shown below is the input/output characteristic of DeadZone with K = 1, Low = 0 and High = 1.

DeadZone Signal Plot



Delay Part



Ideal Time Delay Block (DELAY). Delays the signal for a certain amount of time.

The models and symbols associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please select the appropriate link.

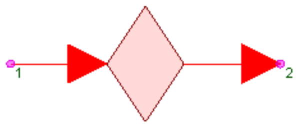
For a list of available parts organized by Schematic Toolbar, please see *Appendix C - Toolbars* (users).

Model
Delay (algorithm)

Symbol
DELAY

For a list of available symbols see the Symbol Reference .

Delay (Delay)



Description: Delay
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Delay Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
N	Sample delay size	1		Integer	NO	[0:∞)
OutputTiming	Output start time: EqualToInput, BeforeInput	EqualToInput		Enumeration	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO

Output Ports

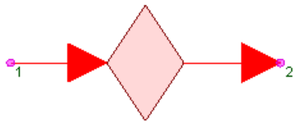
Port	Name	Signal Type	Optional
2	output	anytype	NO

Notes/Equations

1. The Delay model introduces a delay of N samples to the input signal.
2. For every input, there is one output.
3. The initial N output samples have a null value. For scalar signals, a null value is 0. For matrix signals, a null value is a matrix with the same size as the input matrix and with all its elements set to 0.

See:
InitDelay (algorithm)

InitDelay (Delay with Initial Value)



Description: Delay with Initial Value
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *Delay Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
N	Sample delay size	1		Integer	NO	[0:∞)
InitialDelay	Initial data value			None	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim) or *variant* (sim).

Notes/Equations

1. InitDelay delays input tokens from output by N sets of initial delay tokens.
2. For every input, there is an output value.

See:
Delay (algorithm)

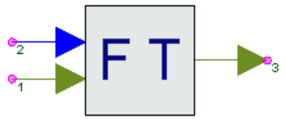
DTFT Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
DTFT (algorithm)	Discrete-Time Fourier Transform

DTFT (Discrete-Time Fourier Transform)



Description: Discrete-Time Fourier Transform
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *DTFT Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
Length	Length of input signal	8		Integer	NO	(0:∞)	L
NumberOfSamples	Number of transform samples to output	128		Integer	NO	(0:∞)	N
TimeBetweenSamples	Time between input samples	1		Float	YES	(0:∞)	T

Input Ports

Port	Name	Description	Signal Type	Optional
1	signal	signal to be transformed	complex	NO
2	omega	frequency values at which to compute the transform	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	dtft	transform values	complex	NO

Notes/Equations

1. The DTFT model calculates the Discrete-Time Fourier transform (DTFT) of the sequence applied at its signal input at each of the frequency points specified on the omega input.
2. At every execution of this model, L samples are read from its signal input, N samples are read from its omega input, and N samples written to its output.
3. The DTFT of a sequence x[n] is a continuous function of ω defined by

$$X(j\omega) = \sum_{n=-\infty}^{\infty} x[n] \times e^{-j\omega n}$$

   If the sequence x[n] is obtained by sampling a continuous time signal $x_c(t)$ at intervals of T, that is $x[n] = x_c(nT)$, and if $X_c(f)$, the continuous-time Fourier Transform of $x_c(t)$, equals 0 for f > 1/(2T), then X(jω) and $X_c(f)$ have the following relationship:

$$X_c(f) = T \times X(j \times 2\pi fT) = T \times \sum_{n=-\infty}^{\infty} x[n] \times e^{-j2\pi fTn}$$

   , for f < 1 / (2T).
4. The DTFT model can calculate *X*(jω) at arbitrary values of ω for sequences *x*[*n*] of finite length. Let the L values on the signal input be *x*[0], *x*[1], ... , *x*[L − 1] and the N values on the omega input be ω[0], ω[1], ... , ω[N − 1]. Then the N values at the output are:

$$X(j\omega[i]) = \sum_{n=0}^{L-1} x[n]e^{-j\omega[i]nT}$$

   , i = 0, 1, ... , N − 1.

Notice that in this last formula the exponent of e has the extra term T compared to the formula defining the DTFT. Therefore, to calculate the Fourier transform of the corresponding continuous time signal $x_c(t)$ at the frequencies f_i, $i = 0, 1, \ldots, N$, generate the values $\omega_i = 2\pi f_i$, apply them at the omega input, and scale the output by T. The values f_i do not need to span the entire frequency range of the signal or be equally spaced.

See:
FFT_Cx (algorithm)

References

1. A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall: Englewood Cliffs, NJ, 1989.

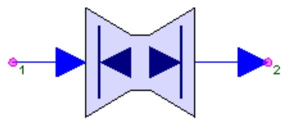
Expand Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Expand (algorithm)	Expander Part of a Compander

Expand (Expander Part of a Compander)



Description: Expander Part of a Compander
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Expand Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
CompressionType	Compression law: MU-law, A-law	MU-law		Enumeration	YES	
CompressionK	Compression constant	1		Float	YES	
Max	Maximum input value magnitude	1		Float	YES	(0.0:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

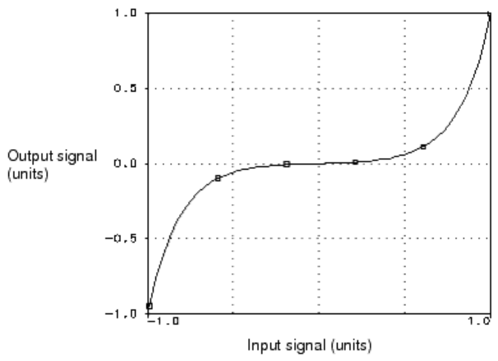
Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. Expand performs *A-law* or *MU-law* expansion.
2. For every input, there is an output value.
3. Let
$$x'(n) = x(n)/V_M$$
then
MU-law:
$$y(n) = \frac{V_M}{\mu}\operatorname{sgn}(x'(n))((1+\mu)^{|x(n)|} - 1)$$
A-law:
$$y(n) = \begin{cases} \frac{V_M(1+ln(A))}{A}x'(n) & \text{for } x'(n) < 1/A \\ \frac{V_M}{A}\operatorname{sgn}(x'(n))e^{(|x'(n)|(1+ln(A))-1)} & \text{for } x'(n) > 1/A \end{cases}$$
where:
x(n) is input for sample n
V_M is Max
y(n) is output for sample n
μ is CompressionK for *MU-Law*
A is CompressionK for *A-Law*
4. Shown below is the input/output characteristic of Expand with Type = *MU-law*, CompressionK = 255 and Max = 1V.

Expand Component Signal Plot



See:
Compress (algorithm)

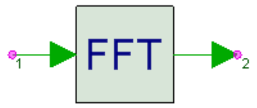
FFT_Cx Part

Categories: *C++ Code Generation* (algorithm), *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
FFT_Cx (algorithm)	Complex Fast Fourier Transform

FFT_Cx (Complex Fast Fourier Transform)



Description: Complex Fast Fourier Transform
Domain: Untimed
C++ Code Generation Support: YES
Associated Parts: *FFT Cx Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
FFTSize	Output transform size	256		Integer	NO
Size	Number of input samples to read	256		Integer	NO
Direction	Direction of transform: Inverse, Forward	Forward		Enumeration	NO
FreqSequence	Sequence for the frequency terms: 0-pos-neg, neg-0-pos	0-pos-neg		Enumeration	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	complex	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	complex	NO

Notes/Equations

1. The FFT_Cx model computes the DFT (Discrete Fourier Transform) of the input signal using a mixed radix FFT (Fast Fourier Transform) algorithm.
2. At every execution of this model, *Size* complex samples are read from the input. This block of *Size* samples is zero padded (if *Size* < *FFTSize*) to create a block of *FFTSize* samples. The block of *FFTSize* samples is then processed by a mixed radix FFT algorithm to produce *FFT_Size* equally spaced samples that is the DFT of the input signal.
3. Direction specifies a forward or inverse FFT.

See:
DTFT (algorithm)

References

1. A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall: Englewood Cliffs, NJ, 1989.

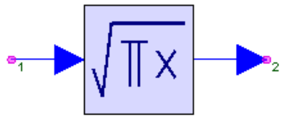
GeometricMean Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
GeometricMean (algorithm)	Geometric Mean Function

GeometricMean (Geometric Mean Function)



Description: Geometric Mean Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *GeometricMean Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
N	Number of samples in a block	10		Integer	NO	[1:∞)
Gain	Gain value	1		Float	YES	(-∞:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. The GeometricMean model computes the geometric mean for every block of *N* input samples.

$$mean[n] = \frac{1}{(n+1)\cdot BlockSize} \cdot \sum_{i=o}^{(n+1)\cdot BlockSize-1} in[i]$$

$$variance[n] = \frac{1}{(n+1)\cdot BlockSize} \cdot \sum_{i=o}^{(n+1)\cdot BlockSize-1} in^2[i] - mean^2[n]$$

2. At every execution of this model, *N* samples are read from the input and 1 sample is written to the output.

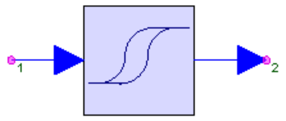
Hysteresis Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Hysteresis (algorithm)	Hysteresis Function

Hysteresis (Hysteresis Function)



Description: Hysteresis Function
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Hysteresis Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Bandwidth	Rate specified damping bandwidth	0	Hz	Float	YES	[0:SampleRate]†
Backlash	Backlash threshold	0		Float	YES	[0:∞)
Gain	Backlash gain	1		Float	YES	(-∞:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. Hysteresis has a internal state which is initialized to zero. The product of this state and the Gain parameter is output.
2. For every input, there is an output value.
3. The state changes only if the magnitude of the difference between the input and the state exceed the Backlash parameter.
4. A fraction of the excess is applied to the state with the sign from the subtraction of the state from the input. This fraction is the Bandwidth parameter normalized by the input rate.
5. A very small bandwidth fraction result in a sluggish hysteresis, i.e. exhibiting lowpass filter behavior, whereas a large bandwidth fraction, e.g. 0.3, result in a stiff hysteresis exhibiting pure backlash behavior.

See:
SchmittTrig (algorithm)

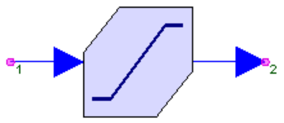
Limit Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Limit (algorithm)	Limiter

Limit (Limiter)



Description: Limiter
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Limit Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
K	Magnitude gain	1		Float	YES	(-∞:0.0) or (0.0:∞)
Bottom	Lower output saturation value	0		Float	YES	(-∞:Top)
Top	Higher output saturation value	1		Float	YES	(-∞:∞)
LimiterType	Type of limiting curve: linear, atan	linear		Enumeration	YES	

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. Limit models two different types of limiting nonlinearities.
2. For every input, there is an output value.
3. If *linear*:

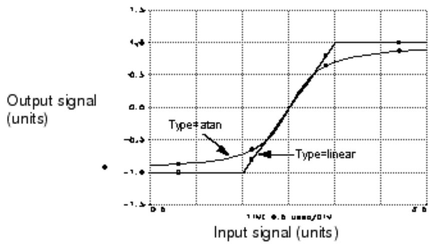
$$y(n)\begin{cases} V_l & if\, x(n) < \frac{V_l}{K} \\ Kx(n) & if \frac{(V_l)}{K} \le x(n) \le \frac{V_h}{K} \\ V_h & if\, x(n) > \frac{V_h}{K} \end{cases}$$

   If *atan*:

$$y(n) = \frac{V_h - V_l}{\pi}\tan^{-1}\left[\frac{4Kx(n) - 2(V_h + V_l)}{V_h - V_l}\right]$$

   where:
   x(n) is input for sample n
   V_l is the Bottom parameter
   V_h is the Top parameter
   K is the K parameter
   y(n) is output for sample n
4. Shown below are input/output characteristics of Limit with parameters K = 1, Bottom = −1, and Top = 1 for *linear* and *atan* types.

Limit Component Signal Plot



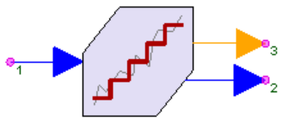
LinearQuantizer Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
LinearQuantizer (algorithm)	Uniform Quantizer with Step Number Output

LinearQuantizer (Uniform Quantizer with Step Number Output)



Description: Uniform Quantizer with Step Number Output
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *LinearQuantizer Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Levels	Number of quantization levels	128		Integer	YES	[2:∞)
Low	Lowest quantization level	-3		Float	YES	(-∞:High)
High	Highest quantization level	3		Float	YES	(-∞:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	amplitude	quantized signal	real	NO
3	stepNumber	quantization level index	int	NO

Notes/Equations

1. The *input* signal is quantized to the number of levels given by the *Levels* parameter.
2. For every input, there is an output at both output ports.
3. Quantization levels start at *Low*, end at *High*, and are uniformly spaced in between.
4. Input is mapped to the closest quantization level.
5. Quantized input is output to the *amplitude* port. Its quantization level index is output to the *stepNumber* port as an integer from 0 (corresponding to the quantization level of *Low*) to *Levels - 1* (corresponding to the quantization level of *High*).

See:
AdaptLinQuant (algorithm)
Quantizer (algorithm)
Quantizer2D (algorithm)

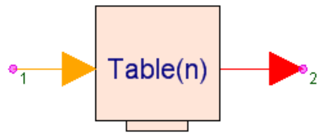
LookUpTable Part

Categories: *C++ Code Generation* (algorithm), *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
LookUpTable (algorithm)	Mapper using Indexed Lookup Table
LookUpTableFxp (hardware)	Fixed Point Look Up Table

LookUpTable (Mapper Using Indexed Lookup Table)



Description: Mapper using Indexed Lookup Table
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *LookUpTable Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Values	Table of output values	[-1; 1]		None	NO

Input Ports

Port	Name	Signal Type	Optional
1	input	int	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim) or *variant* (sim).

Notes/Equations

1. The LookUpTable model implements a lookup table indexed by an integer-valued input.
2. For every input, there is an output value.
3. The input must lie between 0 and $N - 1$, inclusive, where N is the size of the table (size of first dimension of the *Values* parameter). If the *Values* parameter is a multi-dimensional array, then the output is going to be an array whose number of dimensions is one less than the number of dimensions of the *Values* parameter. The first element (or slice across the first dimension) of the *Values* array is indexed by a zero-valued input. An error occurs if the input value is out of the array bounds.
4. For details on creating arrays of data for parameter values, refer to Multidimensional Arrays.

MaxMin Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
MaxMin (algorithm)	Maximum or Minimum Value Function

MaxMin (Maximum or Minimum Value Function)

Description: Maximum or Minimum Value Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *MaxMin Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
N	Number of samples	10		Integer	NO	[0:∞)
MaxOrMin	Output value: min, max	max		Enumeration	YES	
Compare	Compare input value or magnitude: valueIn, magnitudeIn	valueIn		Enumeration	YES	
OutputType	Output value or magnitude: valueOut, magnitudeOut	valueOut		Enumeration	YES	

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	real	NO
3	index	int	NO

Notes/Equations

1. MaxMin finds the minimum or minimum value or magnitude of N input values.
2. For every N input values, one value is output.
3. If Compare = *valueIn*, the input with the maximum or minimum value is located, otherwise the input with the maximum or minimum magnitude is located.
4. If OutputType = *magnitudeOut*, the magnitude of the result is output, otherwise the result is output.
5. The default parameters output the maximum value among N=10 input samples. The index of the maximum is output to the index port where 0 locates the first input value.
6. Use MaxMin to operate over multiple data streams by preceding it with a Commutator and setting the N parameter appropriately.

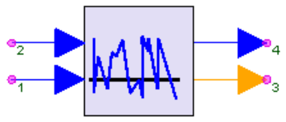
PattMatch Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
PattMatch (algorithm)	Pattern Cross Correlator using Template

PattMatch (Pattern Cross Correlator using Template)



Description: Pattern Cross Correlator using Template
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *PattMatch Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
TempSize	Number of samples in template	32		Integer	NO	(0:∞)
WinSize	Number of samples in search template	176		Integer	NO	[TempSize:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	templ	template input	real	NO
2	window	window input	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
3	index	index output	int	NO
4	values	cross-correlation output	real	NO

Notes/Equations

1. The PattMatch model accepts a template and a search window and tries to find the position in the search window where the template matches best.
2. At every execution of this model, *TempSize* samples are read from the *templ* input and *WinSize* samples are read from the *window* input. At the same time, one sample is written to the *index* output and (*WinSize* − *TempSize* + 1) samples are written to the *values* output.
3. The algorithm for finding the best template match position starts by placing the template at the left end of the window (first samples of template and window are aligned) and calculating the cross-correlation between them. Then the template is shifted across the window one sample at a time and the cross-correlation is computed at each step until the template reaches the right end of the window (last samples of template and window are aligned). The cross-correlation values are output on the *values* output. The *index* output is the value of the shift (in number of samples) that gives the largest cross-correlation.
4. The cross-correlation values are normalized against the energy of the window under the template:
$$C(n) = \frac{\sum_{m=0}^{T_{size}-1} T(m)W(m+n)}{\sum_{m=0}^{T_{size}-1} W(m+n)W(m+n)}$$
where *T* is the template, *W* is the window, *n* is the index value, and T_{size} equals TempSize.
Note that if the template is identical to a certain segment of the window, then the cross-correlation value *C(n)* for that segment will be 1.0. Therefore, the index with

the highest cross-correlation value may not be the best match if that value is greater than 1.0.

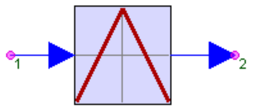
PcwzLinear Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
PcwzLinear (algorithm)	Piecewise Linear Mapper

PcwzLinear (Piecewise Linear Mapper)



Description: Piecewise Linear Mapper
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *PcwzLinear Part* (algorithm)

Model Parameters

Name	**Description**	**Default**	**Units**	**Type**	**Runtime Tunable**
Breakpoints	Endpoints and breakpoints in the mapping	[-1-j, j, 1-j]		Complex array	NO

Input Ports

Port	**Name**	**Description**	**Signal Type**	**Optional**
1	input	input signal	real	NO

Output Ports

Port	**Name**	**Description**	**Signal Type**	**Optional**
2	output	output signal	real	NO

Notes/Equations

1. PcwzLinear implements a piecewise linear mapping from the input to the output.
2. For every input, there is one output value.
3. Mapping is given by a sequence of (x,y) pairs that specify breakpoints in the function. The (x,y) pairs are expressed in the Breakpoints parameter in the complex form, i.e. x+jy. The sequence of x values must be increasing.
4. The function implemented by PcwzLinear can be represented by drawing straight lines between the (x,y) pairs in sequence. Additionally, the first breakpoint is extended to $x=-\infty$, and the last breakpoint is extended to $x=+\infty$. Each input is treated as a point on the x axis, and the corresponding y value is output.
5. For example, the default mapping is the *tent* map. Inputs between -1.0 and 0.0 are linearly mapped into the range -1.0 to 1.0. Inputs between 0.0 and 1.0 are mapped into the same range, but with opposite slope, 1.0 to -1.0. If the input is -2.0, the output will be -1.0. If the input is +2.0, the output will again be -1.0.

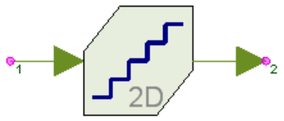
Quantizer2D Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Quantizer2D (algorithm)	2-Dimensional Quantizer using Threshold List

Quantizer2D (2-Dimensional Quantizer using Threshold List)



Description: 2-Dimensional Quantizer using Threshold List
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Quantizer2D Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
VxMax	Maximum real output level	1		Float	YES	(-∞:∞)
VxMin	Minimum real output level	-1		Float	YES	(-∞:VxMax)
Nx	Number of real output levels	16		Integer	YES	[1:∞)
VyMax	Maximum imaginary output level	1		Float	YES	(-∞:∞)
VyMin	Minimum imaginary output level	-1		Float	YES	(-∞:VyMax)
Ny	Number of imaginary output levels	16		Integer	YES	[1:∞)
QuantList	User-defined quantization points			Complex array	NO	

Input Ports

Port	Name	Signal Type	Optional
1	input	complex	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	complex	NO

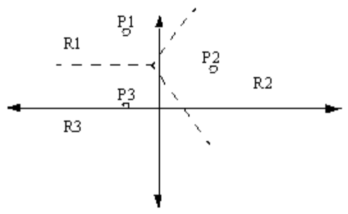
Notes/Equations

1. The complex number input is mapped to one of a finite set of complex numbers.
2. For every input, one value is output.
3. An arbitrary set of output points can be specified with the QuantList parameter. Otherwise, the parameters VxMax, VxMin, Nx, VyMax, VyMin and Ny can be used to create a rectangular grid of output points.
4. Each input is mapped to the nearest output point, where the metric used to determine the nearest output point is the Euclidean distance. This type of a quantizer is also referred to as a Voronoi or a nearest neighbor vector quantizer.
5. 2D Quantizer with Three Output Points shows an example where three output points P1, P2, and P3 have been specified. The entire 2D plane is then divided into 3 regions, R1, R2, and R3, which are shown by the dotted lines. Any input point in region R1 is mapped to the output point P1 (and similarly for the other regions).
6. 2D Quantizer with Output Points On a Grid illustrates how a rectangular grid of output points can be created by using the parameters VxMax, VxMin, Nx, VyMax, VyMin and Ny.
   Due to the regular lattice structure of this quantizer, it can be implemented efficiently in terms of speed. Therefore, it is more efficient to use this second method of specifying a quantizer than using a list of output points.
7. When a list is used to specify output points, data is entered for the QuantList parameter as array of complex values. Data entered as an explicit array has the form:
   QuantList = [1, 0.707+0.707j, j, −0.707+0.707j, −1, −0.707−0.707j, −j, 0.707−0.707j]

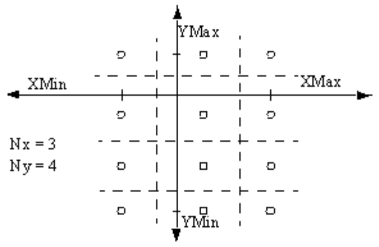
The above list can be used to create a quantizer for an 8PSK receiver whose signal set consists of 8 points equally spaced on a unit circle. Quantizer2D shows the points and the decision regions (in dotted lines) for this quantizer.

See:
AdaptLinQuant (algorithm)
LinearQuantizer (algorithm)
Quantizer (algorithm)

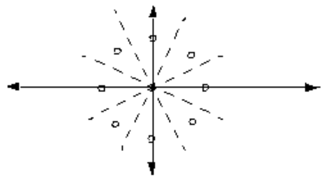
2D Quantizer with Three Output Points



2D Quantizer with Output Points On a Grid



Quantizer2D



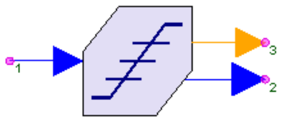
Quantizer Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Quantizer (algorithm)	Quantizer using Threshold List

Quantizer (Quantizer using Threshold List)



Description: Quantizer using Threshold List
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Quantizer Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
Thresholds	Quantization thresholds (increasing order)	[0]		Floating point array	NO
Levels	Output levels (if empty use 0, 1, 2, ...)			Floating point array	NO

Input Ports

Port	Name	Description	Signal Type	Optional
1	input		real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output		real	NO
3	stepNumber	Level number of the quantization from 0 to N	int	NO

Notes/Equations

1. Quantize the input to one of N+1 possible output levels found in the Levels parameter using N thresholds from the Thresholds parameter.
2. For every input, one sample is output to both output ports.
3. Thresholds must be ordered in ascending value.
4. For an input less than or equal to the n-th threshold, but larger than all previous thresholds, Levels[n] is output, and the value n−1 is output to the stepNumber port.
5. If the input is less or equal to the first threshold, then output is the first level, Levels[1]. If the input is greater than all thresholds, output is Levels[N+1].
6. If Levels is specified, there must be one more level than thresholds. The default for Levels is 0, 1, 2, ... N.
7. Quantizer takes on the order of log N iterations to find the correct level.

See:
AdaptLinQuant (algorithm)
LinearQuantizer (algorithm)
Quantizer2D (algorithm)

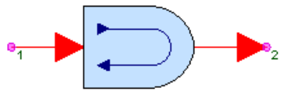
Reverse Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
reverse (algorithm)	Data Reverser

reverse



Description: Data Reverser
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Reverse Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
N	Number of data items read and written	64		Integer	NO	[1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Notes/Equations

1. A block of N samples is input, and that block of N samples is output in the reverse input order.

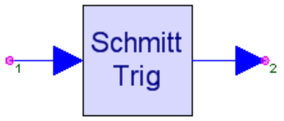
SchmittTrig Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SchmittTrig (algorithm)	Schmitt Trigger

SchmittTrig (Schmitt Trigger)



Description: Schmitt Trigger
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *SchmittTrig Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
ILow	Lower input trigger value	-1		Float	YES	(-∞:IHigh)
IHigh	Higher input trigger value	1		Float	YES	(-∞:∞)
OLow	Lower output trigger value	-1		Float	YES	(-∞:OHigh)
OHigh	Higher output trigger value	1		Float	YES	(-∞:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	real	NO

Output Ports

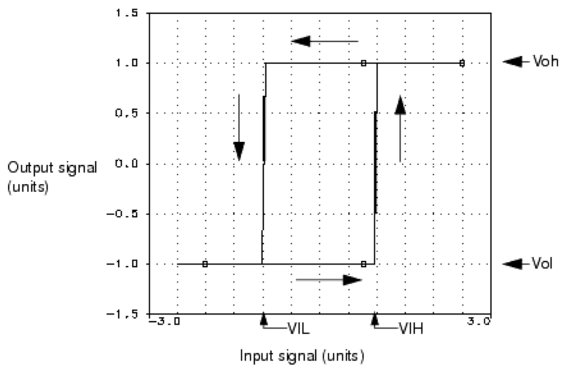
Port	Name	Signal Type	Optional
2	output	real	NO

Notes/Equations

1. SchmittTrig is a Schmitt trigger with programmable levels.
2. For every input, one value is output.
3. The output signal versus input signal plot, with parameters ILow = −1, IHigh = 1, OLow = −1, and OHigh = 1, is shown below.

See:
Hysteresis (algorithm)

SchmittTrig Signal Plot



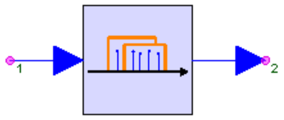
SlidWinAvg Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SlidWinAvg (algorithm)	Sliding-Window Averager

SlidWinAvg (Sliding-Window Averager)



Description: Sliding-Window Averager
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *SlidWinAvg Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
WindowSize	Size of sliding window	3		Integer	NO	(1:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	real	NO

Output Ports

Port	Name	Description	Signal Type	Optional
2	output	output signal	real	NO

Notes/Equations

1. The average of the sample values captured in the most recent window of inputs is output.
2. For every input, an averaged value is output.
3. Samples before the first are zero valued.

See:
Integrator (algorithm)
IntegratorCx (algorithm)
IntegratorInt (algorithm)

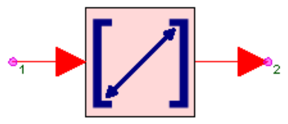
Transpose Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Transpose (algorithm)	Data Transposer

Transpose (Data Transposer)



Description: Data Transposer
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Transpose Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
SamplesInRow	Number of input samples constituting a row	8		Integer	NO	[1:∞)
NumberOfRows	Number of rows in the input matrix	8		Integer	NO	[1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO

Output Ports

Port	Name	Signal Type	Optional
2	output	anytype	NO

Notes/Equations

1. Let M be the NumberOfRows value and N be the SampleInRow value.
2. M × N samples are input in the sequence: $x_{1,1}\ x_{1,2} \cdots x_{1,N}\ ,\ x_{2,1}\ x_{2,2} \cdots x_{2,N}\ , \cdots ,\ x_{M,1}\ x_{M,2} \cdots x_{M,N}$
3. N × M samples are output in transposed order, i.e. in the sequence: $x_{1,1}\ x_{2,1} \cdots x_{M,1}\ ,\ x_{1,2}\ x_{2,2} \cdots x_{M,2}\ , \cdots ,\ x_{1,N}\ x_{2,N} \cdots x_{M,N}$
4. Transpose is a form of interleaver for M input streams of N samples.

See:
reverse (algorithm)

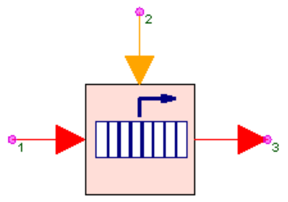
VarDelay Part

Categories: *C++ Code Generation* (algorithm), *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
VarDelay (algorithm)	Variable Delay

VarDelay (Variable Delay)



Description: Variable Delay
Domain: Untimed
C++ Code Generation Support: YES (see Note)
Associated Parts: *VarDelay Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
MaxDelay	Maximum delay	10		Integer	NO	[0:∞)

Input Ports

Port	Name	Signal Type	Optional
1	input	anytype	NO
2	control	int	YES

Output Ports

Port	Name	Signal Type	Optional
3	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *envelope signal* (sim) or *variant* (sim).

Notes/Equations

1. VarDelay introduces a varying delay to the input signal.
2. This model reads 1 sample from the input and writes 1 sample to the output.
3. The delay in samples is controlled by the signal applied to the control port.
4. MaxDelay specifies the maximum control input. A buffer of MaxDelay samples is used to store the most recent samples and is initialized with zero values.
5. The control input determines which sample in the buffer is output. A control value of 0 or less outputs the most recent input. A control value of MaxDelay or greater outputs the oldest sample in the buffer. A control value of N where 0 < N < MaxDelay outputs the sample from the buffer that was read N executions back.
6. This model can be used with the *CrossCorr* (algorithm) model to synchronize signals.

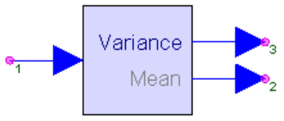
Variance Part

Categories: *Signal Processing* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Variance (algorithm)	Variance Function

Variance (Variance Function)



Description: Variance Function
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *Variance Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
BlockSize	Number of inputs to process between each mean and variance estimate	1		Integer	NO	[1:∞)

Input Ports

Port	Name	Signal Type	Optional
1	in	real	NO

Output Ports

Port	Name	Signal Type	Optional
2	mean	real	NO
3	variance	real	NO

Notes/Equations

1. The Variance model calculates a running estimate of the mean and variance of the input signal. These estimates are reported every *BlockSize* input samples.
2. At every execution of this model, *BlockSize* samples are read from the input and one sample is written to the *mean* and *variance* outputs.
3. The mean and variance values written to the *mean* and *variance* outputs respectively are not the mean and variance of the last block of *BlockSize* samples. They are the mean and variance of all samples read from the beginning of the simulation.

$$mean[n] = \frac{1}{(n+1)\cdot BlockSize} \cdot \sum_{i=0}^{(n+1)\cdot BlockSize-1} in[i]$$

$$variance[n] = \frac{1}{(n+1)\cdot BlockSize} \cdot \sum_{i=0}^{(n+1)\cdot BlockSize-1} in^2[i] - mean^2[n]$$

Sinks

The Sinks library provides parts that can record and/or post-process (perform measurements on) the input they receive. Several commonly used measurements are provided, such as spectrum analysis, EVM, CCDF, BER, etc.

For more information see *About Sinks* (algorithm).

Contents

About Sinks

Access to Envelope Signal Characterization Frequency in a Dataset

When an envelope signal with characterization frequency greater than 0 is saved using a *Sink* (algorithm), the variable <Sink Instance Name>_Fc is saved in the dataset. This is the characterization frequency of the input signal.

PE Estimator Usage

This section explains the use of the probability of error (PE) measurement models *BER_IS* (algorithm) and *BER_FER* (algorithm).

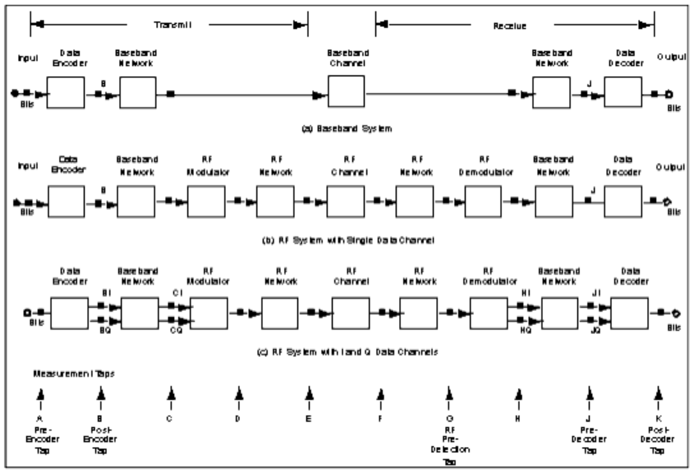
The basic system models and the PE simulation concepts and methodology are explained.

Typical Baseband and RF System Models

Communication systems can be classified broadly as baseband or RF systems. Typically, baseband systems use the PAM data format although other formats, such as pulse width modulation and pulse position modulation, can be used also. RF systems use a wide variety of modulation schemes, such as QAM, PSK schemes (QPSK, DQPSK, PI4DQPSK), and others, such as MSK and FSK. The primary interest is to measure the probability-of-symbol error (Ps) or bit-error rate (BER) of the system.

In the following figure, a general model of a baseband system link is represented in part (a) and general RF system models are represented in parts (b) and (c).

Typical Baseband and RF Systems



An RF system consists of the input bit stream, Transmit Data Encoder, Transmit Baseband Network, RF Modulator, Transmit RF Network, RF Channel, Receive RF Network, RF Demodulator, Receive Baseband Network, Receive Data Decoder, and the output bit stream. The RF system can have a single data channel, or can have I and Q data channels. The Baseband system omits the RF sections. However, the following discussion applies to both systems.

Baseband System Model

The bit stream is often in the NRZ (non-return-to-zero) data format. The Transmit Data Encoder is used for different purposes such as to convert the NRZ data format to another format: for example, multi-level PAM (pulse amplitude modulation) symbol format, or error control coding.

The Baseband Channel has a transmit and receive side and is typically a bandwidth limited environment with intersymbol interference and noise introduced in the channel. The Receive Data Decoder is used to convert the received symbols to the desired output binary data stream.

RF System Model

Often, the bit stream for this system is also in the NRZ data format. The Transmit Data Encoder is used again to convert the NRZ data format to another format or for error control coding. Typically, the Transmit Baseband Network is used to band limit the frequency spectra of the data symbols, and often introduces intersymbol interference to the symbol stream.

The RF modulator can be of many types. Some formats (such as QAM and QPSK) use I and Q data channels while others (such as BPSK) contain a single data channel.

The RF networks and channel include items such as transmit IF and RF filtering, upconverters, high-power amplifiers, coaxial cable, antennas, line-of-sight link, receive RF and IF filters, and receive low-noise amplifier.

The RF Demodulator converts the RF energy back to a baseband symbol stream that includes symbol distortions, interference, and noise. Typically, the Receive Baseband Network is used to filter the received symbol stream to reduce symbol distortion and noise. The Receive Data Decoder is used to convert the received symbols to the desired output bit stream.

PE Measurement Concepts

The probability of error of a system is measured by comparing the output data stream to the input data stream. The BER gives the average number of output bit-errors per input bit; for example, a 10^{-6} BER means that, on the average, 1 output bit-error occurs with 10^{6} input bits.

When the data streams being compared are not simple 2-level (binary) data streams, the measurement is with respect to the data symbols and is called the probability of symbol error (Ps) measurement.

Typical hardware Ps/BER measurements are based on the exact number of symbol/bit-errors and the number of symbols/bits transmitted. This is called a Monte Carlo measurement.

For the PE measurement to be statistically significant, the number of bits transmitted should be much greater than 1/PE (as a rule of thumb).

The relative variance of the PE for N transmitted bits is:

$$VAR = (1 - PE) / (PE \times N)$$

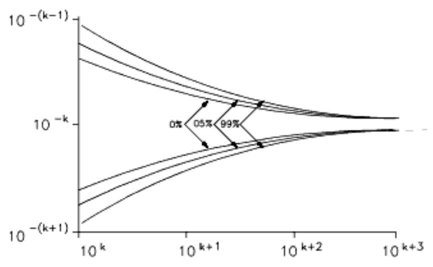
This implies that, for a PE of 10^{-6}, with a relative variance of 0.01, a sample size N of approximately 10^{8} bits is required.

The following figure shows the confidence bands on the PE measurement versus total number of bits observed.

For the program PE measurement, the number of samples required is established by setting the relative variance for the PE measurement. As can be seen, for a PE of 1.0×10^{-k} with 100×10^{k} samples measured, there is a 99% confidence that the actual PE is between 0.77×10^{-k} and 1.3×10^{-k}.

For a low PE uncertainty, a smaller variance is required. However, a smaller variance requires a larger number of transmitted bits.

PE Confidence Bands



Simulation Concepts

Simulation is performed in discrete time steps. At each time step, the signals generated by all the sources are propagated through the system and the outputs are evaluated. Noise can be introduced in the system by means of the random noise sources available. Another method of introducing noise is to use the electrical models of components and to define their noise properties by means of parameters such as noise figure or noise temperature. This is a useful feature that permits you to build an accurate physical model of an actual system. An equivalent noise source is generated automatically that represents the noise properties of the electrical component. No distinction is made between data sources and noise sources during simulation when evaluating the output of the system and the output measured is the net effect due to all sources.

No assumption is made about the nature of the noise at the system output. The statistics of the noise at the output depend on the transformations the noise undergoes when propagating through the system. Thus, the effects of nonlinearities in the system can be simulated accurately.

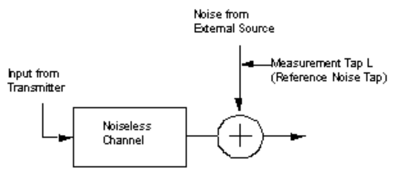
Measurement Taps

The PE performance of a system is calculated by comparing the transmitted data (also referred to as V_{ref}) with the output data of the receiver (referred to as the V_{test}). V_{ref} can be measured at either the pre-encoder tap or the post encoder tap of the transmitter (see the figure Typical Baseband and RF Systems). Correspondingly, V_{test} should be measured at the post-decoder tap or the pre-decoder tap at the receiver. The choice depends on the system under consideration. For example, if the data encoder consists of an NRZ to 4-level PAM data converter, the post-encoder and pre-decoder taps are convenient measurement taps for the reference and test signals, respectively. However the NRZ data bits can be encoded first with an error correction encoder prior to converting these to a 4-level PAM format. Then the data decoder would consist of a 4-level PAM to binary converter followed by an error correction decoder and the PE performance of the entire system could be measured by comparing the pre-encoder signal to the post-decoder signal.

The signal-to-noise ratio of the system cannot be measured by monitoring the receiver output because the output is the combined effect of the data signals and the noise. Therefore, the signal and noise statistics cannot be measured separately. You must either calculate the SNR by examining the structure of the system and the statistical properties of the noise sources, or make other measurements (depending on the system) to determine the SNR.

However, you can introduce noise in the system by placing an external noise source. Typically, such a noise source would be introduced in the channel. If this external noise source is the predominant source of noise in the system, the SNR can be calculated by measuring the power of the noise source and the power of the data signal separately. Then an additional measurement tap (labelled L and called the Reference noise tap) is required as shown in the following figure (also see the figure Typical Baseband and RF Systems).

Introduction of Noise from an External Source into a System


Noise from External Source
Measurement Tap L (Reference Noise Tap)
Input from Transmitter
Noiseless Channel

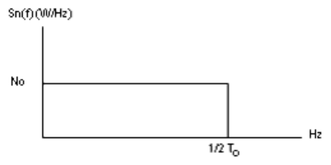
In the preceding figure, the *noiseless channel* block represents effects of the channel such as attenuation of the signal, propagation delay, distortion, and fading. The noise signal measured at the Reference noise tap is called E_{Nref} and the data signal that is used to measure the signal energy is called E_{Sref} . E_{Sref} can be measured at any tap in the system prior to the point where the noise is introduced.

Noise Sources

The nature of the noise introduced in the channel depends on whether the system is a baseband or RF system. Typically, external noise is generated by using the Noise source that is a baseband or RF noise source. The output of the Noise source can be directly added to the signal in the channel.

Another important aspect is the spectral characteristic of the Noise source (see the following figure). Output of the Noise source is a bandlimited white noise signal whose bandwidth is dependent on the time step at which the simulation is carried out. The time interval between two consecutive noise samples T_o , is determined by the parameter TStep in the Noise source. The (baseband) bandwidth of the noise is equal to 1/(2 × TStep) and the power spectral density is constant over this bandwidth.

Spectral Characteristic of the Noise Source



Let

σ = RMS value of the output of the Noise source in volts
T_o = simulation time step in seconds
R_{ref} = default reference resistance is 50 Ohms
$S_n(f)$ = one-sided power spectral density of the bandlimited white noise in W/Hz

Then

$S_n(f) = N_o$ (W/Hz) for $0 \le f \le \frac{1}{2T_o}$ (Hz)

where

$N_o = \frac{2\sigma^2 T_o}{R_{ref}}$

Therefore, the auto-correlation function of the noise is given by

$R_n(\tau) = \frac{N_o}{2T_o}\text{sinc}\left(\frac{\tau}{T_o}\right)$

Because the noise is sampled every T_o seconds, the correlation between the noise samples is given by $R_n(kT_o)$, where k is an integer; therefore, the correlation between the noise samples is zero. In the case of a Gaussian noise source, the samples are also independent.

Relationship between SNR, Es/No, and Eb/No

The signal-to-noise ratio of a system can be calculated in different ways. The ratio of the signal power to the noise power is one such measure and is denoted as the SNR of the system. A measure that is used more commonly is the ratio of the energy per symbol to the power spectral density of the noise (Es/No), or the ratio of the energy per bit to the power spectral density of the noise (Eb/No).

The berMC4 component measures the power in E_{Sref} and E_{Nref} from which the desired signal-to-noise ratio is calculated as follows.

Define the following:

P_{ESref} = power in the E_{Sref} signal in Watts
P_{ENref} = power in the E_{Nref} signal in Watts
T_o = simulation time step in seconds
T_s = symbol time in seconds

Es/No is calculated according to the equations:

$$Es = P_{ESref} \times T_s$$

If the noise signal is in a baseband representation, then

$$No = P_{ENref} \times (2 \times T_o) \text{ and } \frac{Es}{No} = \frac{P_{ESref} \times T_s}{P_{ENref} \times (2 \times T_o)}$$

If the noise signal is in a complex envelope representation, then

$$No = P_{ENref} \times T_o \text{ and } \frac{Es}{No} = \frac{P_{ESref} \times T_s}{P_{ENref} \times T_o}$$

To convert Es/No to Eb/No, let each symbol carry L bits of information. Then Eb/No is simply given by Eb/No = (Es/No)/L.
SNR is simply given by SNR = P_{ESref} / P_{ENref}.

Error Detection

Error detection is performed by sampling V_{ref} and V_{test} every T_s seconds (here T_s is the symbol time) and comparing the samples. If the samples do not lie within the same threshold levels, an error is declared. The ratio of the number of errors counted to the total number of bits transmitted is the estimated PE.

Setting Up BER Simulations

Three important points must be considered when setting up a BER measurement:

- Synchronizing test (V_{test}) and reference (V_{ref}) signals
- Choosing the optimal sampling instant
- Scaling V_{test} and V_{ref} appropriately

Each point is discussed in the following sections. For these discussions, *BER sink* refers to *BER_IS* (algorithm) or *BER_FER* (algorithm).

Synchronizing Test and Reference Signals

A successful BER simulation requires that Vtest and Vref are synchronized. Otherwise, the BER measurement result are most likely to be close to 0.5 (50%). Vtest and Vref can be synchronized in one of two ways: Manual Synchronization and Automatic Synchronization.

- **Manual Synchronization**
  If the exact delay between V_{test} and V_{ref} is known, synchronization can be achieved easily by introducing the same amount of delay in V_{ref}. In this case, set the DelayBound parameter of the BER sink to 0 to turn off the auto synchronization

feature.
The exact delay between V $_{test}$ and V $_{ref}$ can be found in several ways:
  - Sometimes the delay introduced by each component in the path between V $_{test}$ and V $_{ref}$ is known. In this case, adding the delays introduced by each component gives the exact delay between V $_{test}$ and V $_{ref}$.
  - V $_{test}$ and V $_{ref}$ can be recorded (using *Sink* (algorithm)) and plotted. Often, by observing the plots, the delay between the two signals can be determined.
  - Cross-correlation (see *CrossCorr* (algorithm)) can be used to measure the delay between V $_{test}$ and V $_{ref}$.

When determining the delay by observing the plots of V $_{test}$ and V $_{ref}$ versus time or by using cross-correlation, turn off noise and other impairments in the system (set power level of noise sources very low and make amplifiers linear). When these impairments are turned off, the V $_{test}$ waveform is close to the ideal waveform, which helps determine the delay more easily and accurately.

- **Automatic Synchronization**
If the exact delay between V $_{test}$ and V $_{ref}$ is unknown, the auto synchronization feature of the BER sink can be used. In this case, the DelayBound parameter of the BER sink must be set to a value that is an upper bound of the exact delay between the two signals. The BER sink then cross-correlates the two signals and tries to estimate the delay between them. Since the auto synchronization feature relies on cross-correlation to estimate the delay, the BER sink may not be able to always synchronize the two signals, especially at low signal-to-noise ratios.
An upper bound of the delay between V $_{test}$ and V $_{ref}$ can be found using any of the three ways described above used to find the exact delay.
  - Knowing the upper bound of the delay introduced by each of the components in the path between V $_{test}$ and V $_{ref}$ and adding these upper bounds.
  - Observing the plots of V $_{test}$ and V $_{ref}$ versus time.
  - Using cross-correlation (*CrossCorr* (algorithm)) to get an initial estimate of the delay and adding a few simulation time steps to it.

When the auto synchronization feature is used, no delay needs to be added to the reference signal. However, if the upper bound of the delay is large (greater than 50 symbol periods) and if a lower bound of the delay (DL) between V $_{test}$ and V $_{ref}$ is known, then we recommend that V $_{ref}$ is delayed by this amount (DL) and DelayBound is reduced by the same amount (DL). This setting reduces the memory used by the BER sink and speeds up the synchronization process. For example, assume that the delay introduced by the transmitter and receiver filters is 20 msec and the delay introduced by the rest of the components (e.g. RF channel) has an upper bound of 35 msec. One way to set up the BER simulation is to not delay the reference signal at all and set the DelayBound parameter of the BER sink to 55 msec. Another (recommended) way is to delay the reference signal by 20 msec and set DelayBound to 35 msec.

Choosing the Optimal Sampling Instant

The BER sink downsamples V $_{test}$ and V $_{ref}$ to one sample per symbol before it compares them in order to detect errors. The first sample of V $_{ref}$ is taken at the time instant specified by the Start parameter. If auto synchronization is turned off (DelayBound = 0), then the first sample of V $_{test}$ is also taken at Start. If auto synchronization is turned on, then the first sample of V $_{test}$ is taken at Start+Delay, where Delay is the delay the BER sink estimated.

The optimal sampling instant is the instant where there is no ISI (intersymbol interference) or where the minimum ISI occurs. For systems, using root raised cosine filters at the transmitter and receiver, the optimal sampling instant is at the center of the symbol period. Therefore, set the Start parameter of the BER sink as follows:

- if no delay is introduced in the reference signal, set Start to
N x SymbolTime + int((SampPerSym - 1) / 2) x TStep
where N is a positive integer, SymbolTime is the symbol period, SampPerSym is the number of samples per symbol in V $_{test}$ and V $_{ref}$, and TStep is the simulation time step for V $_{test}$ and V $_{ref}$.
- if a delay of D is introduced in the reference signal, set Start to

N x SymbolTime + int((SampPerSym - 1) / 2) x TStep + D

If your system has differential encoding/decoding, synchronization loops, carrier recovery sub-systems, or other sub-systems that need time to reach their steady state, set N to a value that is large enough to enable the entire system to reach steady state.

Scaling of Test and Reference Signals

If the automatic threshold setting is used (BER sink ThresholdSetting parameter set to *automatically*) then the error detection thresholds are set to (2 x i - N) / N, i = 1, 2, ..., N, where N is the number of thresholds (BER sink NumThresholds parameter). NumThresholds must be set to the number of signal levels minus 1 (3 for 4-PAM, 7 for 8-PAM, 3 for 16-QAM (3 levels per axis), 7 for 64-QAM (7 levels per axis)). The expected signal levels are located at the midpoints between the thresholds, that is at (2 x i - 2 - N) / N, i = 1, 2, ... , N + 1. Both V_{test} and V_{ref} must be scaled appropriately so that when these are sampled at the optimal sampling instant and assuming ideal conditions (no noise or other distortions) these generate samples at the expected levels. Turning off the noise and the rest of the impairments in the system and plotting the eye diagrams for Vtest and Vref can help determine the signal levels at the optimal sampling instant and therefore, the appropriate scale factors.

References

1. J. Baprawski, "Generalized Modulation Mathematics Applied to RF System Simulation," *Proceedings of the RF Expo West Conference*, Santa Clara, CA, February 5-7, 1991, pp. 127-147.
2. T. T. Ha, *Digital Satellite Communications*, McGraw-Hill, 1990.
3. N. Kanaglekar, et al. "Wave Analysis of Noise in Interconnected Multiport Networks," *IEEE Transactions on Microwave Theory and Techniques*, Vol. MTT-35, No. 2, February 1987, pp. 112-115.
4. D. Lu and J. Baprawski, "The Quasi-Analytical Method for Estimating Error Probabilities of M-ary Signaling Systems with Intersymbol Interference," *Proceedings of the 13th Symposium on Information Theory and its Applications* , Tateshina, Japan, January 23-25, 1991, pp. 109-114.
5. D. Lu and K. Yao, "Improved Importance Sampling Technique for Efficient Simulation of Digital Communication," *IEEE Journal on Selected Areas in Communication*, J-SAC, Vol. 6, No. 1, January 1988, pp. 67-75.
6. R. W. Lucky, J. Salz and E. J. Weldon, Jr., *Principles of Data Communications*, McGraw-Hill, 1968.
7. P. Z. Peebles, Jr., *Digital Communication Systems*, Prentice-Hall, 1987.
8. G. Proakis, *Digital Communication*, McGraw-Hill, 1989.
9. K. S. Shanmugan, *Digital and Analog Communication Systems*, John Wiley & Sons, 1985.
10. B. Sklar, *Digital Communications*, Prentice-Hall, 1988.

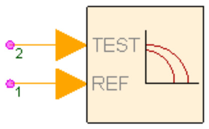
BER_FER Part

Categories: *Sinks* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
BER_FER (algorithm)	Bit and Frame Error Rate Measurement

BER_FER (Bit and Frame Error Rate Measurement)



Description: Bit and Frame Error Rate Measurement
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *BER FER Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
StartStopOption	Sink collection mode: Auto, Samples, Time	Auto		Enumeration	NO	
SampleStart	Sample number to start data collection	0		Integer	NO	[0:∞)
SampleStop	Sample number to stop data collection	Num_Samples - 1		Integer	NO	[SampleStart:∞)
SampleDelayBound	Upper bound of delay between test and ref inputs in number of samples	0		Integer	NO	[0:∞)
TimeStart	Time to start data collection	Start_Time	s	Float	NO	[0:∞)
TimeStop	Time to stop data collection	Stop_Time	s	Float	NO	[TimeStart:∞)
TimeDelayBound	Upper bound of delay between test and ref inputs	0	s	Float	NO	[0:∞)
BitsPerFrame	Bits per frame	100		Integer	NO	[1:∞)
EstRelVariance	BER estimation relative variance	0.01		Float	YES	[0:1)
StatusUpdatePeriod	Status update period in number of bits	1000		Integer	NO	[1:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	ref	reference bit stream	int	NO
2	test	test bit stream	int	NO

Notes/Equations

1. The BER_FER model can be used to measure the BER (bit error rate) and FER (frame error rate) of a system. In some systems, FER is referred to as PER (packet error rate) or BLER (block error rate). The input signals to the reference (REF) and test (TEST) inputs must be bit streams. The bit streams must be synchronized, otherwise the BER/FER estimates are wrong.
2. The Start parameter defines when data processing starts. The end of data processing depends on the settings of the Stop and EstRelVariance parameters:
   - If EstRelVariance is 0.0, then data processing ends when Stop is reached.
   - If EstRelVariance is greater than 0.0, then data processing ends when EstRelvariance is met or when Stop is reached. In this case, Stop acts as an upper bound on how long the simulation runs just in case the simulation takes too long for EstRelVariance to be met. In this mode of operation, messages are printed in the simulation log showing the value of estimation relative variance as the simulation progresses. The EstRelVariance parameter can be used to control the quality of the BER estimate obtained. The lower the value of EstRelVariance the more accurate the estimate is.
     For more details, refer to *PE Measurement Concepts* (algorithm). Note that the equation for the estimation relative variance described in this section assumes that the errors happen randomly (as in the case of an AWGN channel) and not in bursts (as in the case of a fading channel).

3. The BitsPerFrame parameter sets the number of bits in each frame. A frame is considered to be in error if at least one of the bits in the frame is detected incorrectly. If the bit errors are independent identically distributed events then BER and FER are related through the equation $\mathrm{FER} = 1 - (1 - \mathrm{BER})^{\mathrm{BitsPerFrame}}$.
   To estimate BER/FER over an exact number of frames set EstRelVariance to 0.0 and Stop to Start + N × BitsPerFrame − 1, where Start is the value of the Start parameter, BitsPerFrame is the value of the BitsPerFrame parameter and N is the number of frames to be simulated.
4. The StatusUpdatePeriod parameter can be used to control how often estimation relative variance status messages are reported to the simulation log.
5. For the theoretical BER expressions for commonly used modulation formats see *Theoretical BER Curves* (algorithm).

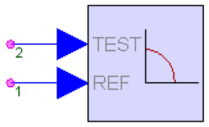
BER_IS Part

Categories: *Sinks* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
BER_IS (algorithm)	Error Probability Measurement using Improved Importance Sampling

BER_IS (Error Probability Measurement using Improved Importance Sampling)



Description: Error Probability Measurement using Improved Importance Sampling
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *BER IS Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Start	Start time for data collection	Start_Time	s	Float	NO	[0:∞)
Stop	Stop time for data collection when EstRelVariance is not met	Stop_Time	s	Float	NO	(Start:∞)
EstRelVariance	Estimation relative variance	0.01		Float	NO	(0:1]
DelayBound	Upper bound of delay for synchronizing inputs (synchronizer is turned off when DelayBound = 0.0)	0.0	s	Float	NO	[0:∞)
SymbolTime	Symbol time	10e-6	s	Float	NO	[TStep:∞)†
SystemType	System type: PAM, QAM, QPSK, DQPSK, PI4DQPSK	PAM		Enumeration	NO	
NumThresholds	Number of thresholds for the error detection	1		Integer	NO	[1:∞)
ThresholdSetting	Threshold setting option: automatic, manual	automatic		Enumeration	NO	
Thresholds	Threshold values	[0]		Floating point array	NO	
NoiseBandwidth	Noise bandwidth	50e3	Hz	Float	NO	(0:∞)
SNR_Option	SNR option: Es/No, Eb/No	Es/No		Enumeration	NO	
SNR_Start	Start value for SNR sweep	5		Float	NO	(-∞:∞)
SNR_Step	Step value for SNR sweep	1		Float	NO	(-∞:∞)
SNR_NumSteps	Number of steps for SNR sweep	5		Integer	NO	[0:∞)
StatusUpdatePeriod	Status update period in number of symbols	100		Integer	NO	[1:∞)

Input Ports

Port	Name	Description	Signal Type	Optional
1	reference	reference data input	real	NO
2	test	test signal input	real	NO

Notes/Equations

1. The BER_IS model measures the probability of error based on the Improved Importance Sampling (IIS) method [1 - 3], which can quickly estimate error probabilities for PAM, QAM, QPSK, DQPSK, and π/4-DQPSK systems. In order to reduce simulation time, error events are made to occur more frequently by modifying the probability density function of the noise. To compensate for this, a weighting function is used to adjust the error probability so that the resulting estimate is unbiased.
   Note that the IIS method is system dependent and so BER_IS cannot be used to estimate error probability for other systems.
2. The input1 signal is the reference data input and should have no distortion or intersymbol interference. The input2 signal is the one against which the BER measurement is made.
3. The Start parameter defines when data processing starts. The end of data processing depends on the settings of the Stop and EstRelVariance parameters. Data processing ends when EstRelvariance is met or when Stop is reached, that is, Stop acts as an

upper bound on how long the simulation runs just in case the simulation takes too long for EstRelVariance to be met.

4. The EstRelVariance parameter is used to control the quality of the BER estimate obtained. The lower the value of EstRelVariance the more accurate the estimate is and the longer the simulation runs. For more details, refer to *PE Measurement Concepts* (algorithm). Note that the equation for the estimation relative variance described in this section assumes that the errors happen randomly (as in the case of an AWGN channel) and not in bursts (as in the case of a fading channel).
5. The Start parameter also defines the first sampling instant for the test and reference signals and therefore needs to be set so that the signals are sampled at the optimal, zero ISI (InterSymbol Interference) sampling instant. This is typically the center of the reference signal symbol period.
    - If the test and reference signals are already synchronized (DelayBound = 0) then they are sampled every SymbolTime sec starting at Start.
    - If the test and reference signals are not synchronized (DelayBound parameter is greater than 0) then the test signal is first synchronized to the reference signal by cross correlating them to estimate the delay between them. Then the reference signal is sampled every SymbolTime sec starting at Start and the test signal is sampled every SymbolTime sec starting at Start + Delay, where Delay is the estimated delay between test and reference.
6. The DelayBound parameter specifies an upper bound for the delay between the test and reference signals. If DelayBound is set to 0, then the test and reference signals are assumed to be synchronized and no synchronization is performed. If DelayBound > 0, then cross correlation is performed between the test and reference signals to determine the delay between them. In this case, DelayBound is used as the maximum lag at which the cross correlation is computed. If the actual delay between test and reference is bigger than DelayBound, the simulation results will be unexpected.
7. The SymbolTime parameter defines the symbol duration and is used to sample the input signals once in every symbol period.
8. The SystemType parameter defines the modulation type used in the system. As mention earlier, the IIS method is system dependent.
9. When SystemType is PAM or QAM then the number of detection thresholds needs to be specified in the NumThresholds parameter. NumThresholds is one less than the number of levels. For example, for a PAM-4 system, there are 4 levels and 3 thresholds. For a 64-QAM system, there are 8 levels (on each axis) and 7 thresholds.
10. The ThresholdSetting parameter defines how the detection thresholds are set.
    - If ThresholdSetting is set to automatic, then the thresholds are set at equal distance between the ideal signal levels, which are assumed to be equally spaced between -1 and 1. For example, for a PAM-4 system, the ideal levels are -1, -1/3, 1/3, 1 and so the thresholds are set to -2/3, 0, 2/3.
    - If ThresholdSetting is set to manual, then the thresholds need to be specified in the Thresholds array parameter. The values of the Thresholds array must be between -1 and 1 and they should be sorted in increasing order.
      In either case, the reference signal needs to be scaled so that when downsampled (once every symbols period) it is in the range [-1, 1]. The test signal also needs to be scaled so that when downsampled (once every symbols period) it is in the range [-1, 1] if there is no distortion (ISI, distortion from non-linearities, etc). If distortion is present then the downsampled test signal will be slightly outside the range [-1, 1] but this should be due only to distortion and not to gain (signal amplification/attenuation) factors between the point of the reference generation and the point of detection.
11. The BER_IS model adds white Gaussian noise to the test signal (there is no need for an external noise source). The NoiseBandwidth parameter is used in the calculation of the noise variance. To get the theoretical results in an AWGN (Additive White Gaussian Noise) environment, set NoiseBandwidth to one half of the symbol rate (1 / (2 × SymbolTime)). The noise variance also depends on the SNR value. Since the BER_IS model adds noise to the signal internally it can reuse the same input signals and sweep the noise power to generate an entire BER waterfall curve. The SNR_Start, SNR_Stop, and SNR_Step parameters are used to set this internal sweep. The SNR_Option parameter specifies whether the SNR values are E_s / N_o or E_b / N_o.
12. The StatusUpdatePeriod parameter controls how often status messages are printed to the simulation log. These messages show the value of the estimation relative variance for the highest SNR as the simulation progresses. In addition, to these messages every time the estimation relative variance is met for a specific SNR value, a message is printed in the simulation log and that SNR value is removed from the internal SNR sweep. This parameter does not affect the simulation results. It only controls the frequency of the messages in the simulation log.
13. Symbol Error Probability and Bit Error Probability
    The BER_IS model can measure the symbol error probability or symbol error rate P_{se} for a single channel. For a system with only one channel, such as PAM, the bit error probability or bit error rate P_{be} can be calculated by $P_{be} = P_{se} / L$, where L is the number of bits per symbol (for example, for a PAM-8 system L = 3). Note that the above formula assumes that Gray coding was used to map the bits to symbols and that symbol errors occur only between adjacent symbols, so that a symbol error translates to one bit error.

For systems with I and Q channels, such as QAM, QPSK, π/4-DQPSK, two BER_IS models must be used to measure the symbol error rate for each channel, P_{seI} and P_{seQ}. Then P_{se} for the whole system can be calculated by $P_{se} = P_{seI} + P_{seQ} - P_{seI} \times P_{seQ}$.

To calculate the bit error rate P_{be} for a system with two channels, first calculate the bit error rate for each channel P_{beI} and P_{beQ} using the equations $P_{beI} = P_{seI} / L$ and $P_{beQ} = P_{seQ} / L$, where L is the number of bits per symbol in each channel (for example, for QPSK $L = 1$, for 16-QAM $L = 2$, for 64-QAM $L = 3$). Again, the assumptions that Gray coding was used to map the bits to symbols and that symbol errors occur only between adjacent symbols are made. After P_{beI} and P_{beQ} have been calculated, the bit error rate for the whole system is given by $P_{be} = (P_{beI} + P_{beQ}) / 2$.

14. Simulation Time Improvement with IIS Simulation
    For a QAM system with a bit error rate of 10^{-6}, a Monte Carlo simulation requires approximately 10^{8} bits for a reasonable accuracy (estimation relative variance of 0.01). For the same system and accuracy an IIS simulation requires only 800 bits [1].
15. Here are some steps to follow in order to set up a successful BER simulation using BER_IS:
    - Scale the test and reference signals appropriately as explained in note 9.
    - Decide whether you will synchronize the test and reference signals manually or you will use the synchronization feature of the BER_IS model.
    - If you decide to synchronize the test and reference signals manually
        - determine the delay between them (this can be done by using the CrossCorr model or by plotting the two signals and visually estimating the delay)
        - delay (using the Delay model) the reference signal by the delay determined
        - connect the delayed reference signal to the reference input
        - connect the test signal to the test input
        - set the DelayBound parameter to 0
    - If you decide to use the synchronization feature of the BER_IS model
        - determine an upper bound of the delay between the test and reference signals (this can be done by using the CrossCorr model or by plotting the two signals and visually estimating the delay)
        - connect the reference signal to the reference input
        - connect the test signal to the test input
        - set the DelayBound parameter to the upper bound of the delay you determined
    - Set the Start parameter to the center of the symbol period of the reference signal. If you have manually synchronized the test and reference signals by adding delay in the reference signal make sure you account for this delay when setting Start.
    - Set the Stop parameter to an upper bound of how much time you want to simulate.
    - Set the EstRelVariance parameter to the desired value for the accuracy you want.
    - Set SymbolTime, SystemType, and NumThresholds based on the system you want to simulate.
    - Decide whether to use automatic or manual threshold setting and set the ThresholdSetting and Thresholds parameters accordingly.
    - Set the NoiseBandwidth parameter to the noise bandwidth of your system. For theoretical results in an AWGN environment NoiseBandwidth must be set to one half of the symbol rate (1 / (2 × SymbolTime)).
    - Set SNR_Option, SNR_Start, SNR_Step, and SNR_NumSteps to the appropriate values that will generate the BER waterfall curve you want.
    - Set StatusUpdatePeriod to the desired value.
16. For the theoretical BER expressions for commonly used modulation formats see *Theoretical BER Curves* (algorithm).
17. For general information regarding sinks, refer to *About Sinks* (algorithm).

References

1. D. Lu and K. Yao, "Improved Importance Sampling Technique for Efficient Simulation of Digital Communication Systems," *IEEE J. Select. Areas Commun*. vol. 6, pp. 67-75, Jan. 1988.
2. D. Lu and K. Yao, "Estimation Variance Bounds of Importance Sampling Simulations in Digital Communication Systems," *IEEE Trans. Commun*. vol 39, pp. 1413-1417, Oct. 1991.
3. J. Chen, D. Lu, J. Sadowski, and K. Yao "On Importance Sampling in Digital Communications - Part I: Fundamentals," *IEEE J. Select. Areas in Commun*. vol 11. No. 3, pp. 289-299, April, 1993.

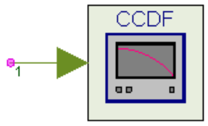
CCDF Part

Categories: *Sinks* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
CCDF_Env (algorithm)	Complementary Cumulative Distribution Function (CCDF)
CCDF_Cx (algorithm)	Complementary Cumulative Distribution Function (CCDF)

CCDF_Cx



Description: Complementary Cumulative Distribution Function (CCDF)
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *CCDF Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Start	Start time for data collection	Start_Time	s	Float	NO	[0:∞)
Stop	Stop time for data collection	Stop_Time	s	Float	NO	[Start:∞)
NumBins	Number of points in the CCDF curve	100		Integer	NO	[3:65535]
OutputPeakMean	Output signal peak and mean values: NO, YES	NO		Enumeration	NO	

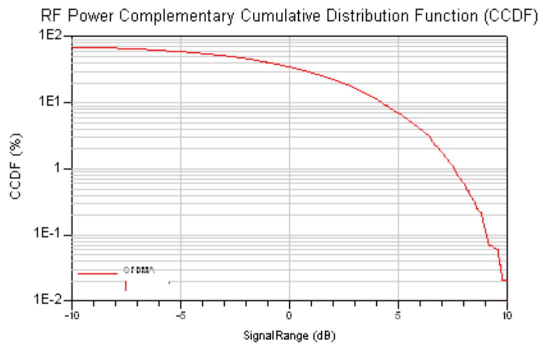
Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	complex	NO

Notes/Equations

1. The CCDF_Cx model computes the complementary cumulative distribution function (CCDF) of a complex signal. As its name suggests, CCDF is the complement of the cumulative distribution function (CDF). The relationship between CCDF and CDF is CCDF = 1 - CDF.
   The calculation of the CCDF is described below:
   - Calculate the RMS value for all measured samples; this becomes the 0 dB point on the x-axis.
   - Normalize all samples to the RMS value in units of dB.
   - Split the x-axis in equal width NumBin bins starting from minimum measured power to maximum measured power.
   - Determine which x-axis bin each sample belongs to.
   - Calculate the total number of samples that are greater than or equal to each x-axis bin and output it as a percent of the number of samples measured.
2. In addition to the CCDF measurement, this model can provide PeakPower (the peak power for the input signal) and MeanPower (the mean or average power of the input signal). To compute and output MeanPower and PeakPower, set the parameter OutputPeakMean to YES.
3. The CCDF measurement is a very common measurement performed on 2G, 3G and 4G wireless signals. The CCDF curve shows the probability that the instantaneous signal power will be higher than the average signal power by a certain amount of dB. The independent axis of the CCDF curve shows power levels in dB with respect to the average signal power level (0 dB corresponds to the average signal power level). The dependent axis of the CCDF curve shows the probability that the instantaneous signal power exceeds the corresponding power level on the independent axis. The following figure shows the CCDF curve for a WiMax 802.16e Downlink signal. In the figure, you can see that the instantaneous signal power exceeds the average signal power (0 dB) for 35% of the time. You can also see that the instantaneous signal power exceeds the average signal power by 5 dB for only 7% of the time.

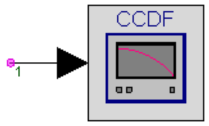
CCDF measurement for a WiMax 802.16e Downlink Signal



4. The amount of data collected and on which the CCDF measurement is performed is controlled by the Start and Stop parameters. Data collection always begins at the time instant specified by Start and ends at the time instant specified by Stop.
5. In most wireless communication systems with framed or burst data, the CCDF measurement must be performed on the active part of the signal (idle parts between frames/burst should be excluded).
6. For general information regarding sinks, refer to *About Sinks* (algorithm).

See:
CCDF_Env (algorithm)

CCDF_Env



Description: Complementary Cumulative Distribution Function (CCDF)
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *CCDF Part* (algorithm)

Model Parameters

Name	**Description**	**Default**	**Units**	**Type**	**Runtime Tunable**	**Range**
Start	Start time for data collection	Start_Time	s	Float	NO	[0:∞)
Stop	Stop time for data collection	Stop_Time	s	Float	NO	[Start:∞)
NumBins	Number of points in the CCDF curve	100		Integer	NO	[3:65535]
OutputPeakMean	Output signal peak and mean values: NO, YES	NO		Enumeration	NO	

Input Ports

Port	**Name**	**Description**	**Signal Type**	**Optional**
1	input	input signal	envelope	NO

Notes/Equations

1. The CCDF_Env model computes the complementary cumulative distribution function (CCDF) of an envelope signal. For more details on the CCDF measurement see *CCDF_Cx* (algorithm).
2. For general information regarding sinks, refer to *About Sinks* (algorithm).

See:
CCDF_Cx (algorithm)

DataPort Part

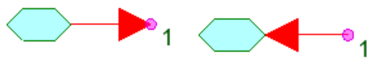
Terminal: Standard Data Port Terminal

Categories: *Sinks* (algorithm), *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model
DataPort (algorithm)

DataPort (Standard Data Port Terminal)



Description:
Category:
Domain:
Associated Part: *DataPort Part* (algorithm)

Parameters

Name	Description	Default	Symbol	Unit	Type	Range
PORT	Port number	1			int	[0, ∞)
Direction	Data direction: Input, Output, (none)	Input			enum	
Data Type	Data type: Any Type, Fixed Point, Floating Point (Real), Integer, Complex, Envelope Signal, Variant, Floating Point (Real) Matrix, Integer Matrix, Complex Matrix	Any Type			enum	
Bus	Is this a bus: NO, YES	NO			enum	
Commutative	Bus ordering matters: NO, YES	NO			enum	
Optional	Is this port optional: NO, YES	NO			enum	
ZO	Impedance	50		Ohm	real	(-∞, ∞)

Port

Port	Name	Description	Signal Type	Optional
settable	settable	Schematic Data Port	settable	settable

Notes/Equations

1. DataPort is a port only model that defines an input or an output for a subnetwork.
2. For any number of DataPort parts in one subnetwork, one DataPort must have a PORT parameter set to 1.
3. Data flow subnetworks uses only *Input* or *Output* for the Direction parameter.
4. For further information on buses, see *Connection Terminology* (users).
5. If this port connects to a bus and Commutative is *NO*, connection line ordering must be maintained.
6. If Optional is *NO*, this port must be connected in a simulation.
7. ZO is the reference impedance for the port. Data flow subnetworks do not use this parameter.

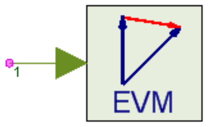
EVM Part

Categories: *Sinks* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
EVM_Env (algorithm)	Error Vector Magnitude Measurement
EVM_Cx (algorithm)	Error Vector Magnitude Measurement

EVM_Cx



Description: Error Vector Magnitude Measurement
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *EVM Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Start	Start time for data collection	Start_Time	s	Float	NO	[0:∞)
SymTime	Symbol duration time	10e-6	s	Float	NO	[TStep:1000*TStep]†
SymBurstLen	Burst length in number of symbols	100		Integer	NO	[1:∞)
MeasType	Measurement type: EVM RMS, EVM Peak, C0, Frequency error, Droop, Error sequence, Magnitude error sequence, Phase error sequence, Sampled data	EVM RMS		Enumeration	NO	
ModType	Modulation type: BPSK, QPSK, HPSK, PI/4 DQPSK, PSK8, PSK16, QAM4, QAM16, QAM32, QAM64, QAM128, QAM256, PAM4, PAM8, User defined	QPSK		Enumeration	NO	
Constellation	Complex constellation values	[1+j, 1-j, -1-j, -1+j]		Complex array	NO	
OptimizeSamplingInstant	Automatically find optimal sampling instant: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	complex	NO

Notes/Equations

1. The EVM_Cx model is used to perform an EVM (Error Vector Magnitude) measurement for a complex signal. EVM measurements are used to evaluate the modulation accuracy of modulators. It is used, for example, in the IS-54 TDMA digital cellular to set the minimum specifications for modulation accuracy of π/4-DQPSK modulators.
The defining equations for the EVM measurement follow the definition in the *EIA/TIA IS-54-B TDMA Cellular System Dual-Mode Mobile Station-Base Station Compatibility Standard, section 2.1.3.3.1.3.3 (Error Vector Magnitude Requirement).* Let Z(k) denote the actual complex vectors (I and Q) produced by observing the real transmitter through an ideal receiver filter at instants k, one symbol period apart. S(k) is defined as the ideal reference symbol (normalized such that its maximum energy symbol falls on the unit circle). Then, Z(k) is modeled as:
$Z(k) = [C_0 + C_1(S(k) + E(k))]W^k$
where

   W = e^{Dr+jDa} , accounts for both a frequency offset (Da radians/symbol phase rotation) and an amplitude change rate (of Dr nepers/symbol)

   C_0 is a complex constant origin offset (in Volts)

   C_1 is a unitless complex constant representing the arbitrary phase and output power of the transmitter, and

E(k) is the residual vector error on sample S(k) (in Volts)

The sum square error vector is

$$\sum_{k=0}^{N-1} |E(k)|^2 = \sum_{k=0}^{N-1} \left| \frac{[Z(k)W^{-k} - C_0]}{C_1} - S(k) \right|^2$$

where N is equal to SymBurstLen and C $_0$, C $_1$, W are chosen such as to minimize the above expression.

EVM (rms) is defined to be the rms value of | E(k) | normalized by the rms value of | S(k) |. Therefore,

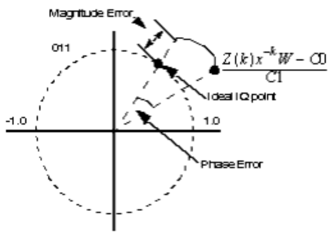
$$EVM(rms) = \frac{\sqrt{\frac{1}{N} \cdot \sum_{k=0}^{N-1} |E(k)|^2}}{\sqrt{\frac{1}{N} \cdot \sum_{k=0}^{N-1} |S(k)|^2}} = \sqrt{\frac{\sum_{k=0}^{N-1} |E(k)|^2}{\sum_{k=0}^{N-1} |S(k)|^2}}$$

The symbol EVM at symbol k is defined as

$$EVM(k) = \frac{|E(k)|}{\sqrt{\frac{1}{N} \cdot \sum_{k=0}^{N-1} |S(k)|^2}}$$

which is the vector error magnitude at symbol k normalized by the rms value of | S(k) |.

2. The MeasType parameter determines the output of the component.
   - If MeasType = EVM RMS, the output is the EVM(rms) value as defined in the equations of note 1. This value is not in percent.To get EVM RMS in percent, multiply by 100.
   - If MeasType = EVM Peak, the output is max{ EVM(k) }, where the maximum is evaluated over k = 0, 1, ... , N-1. This value is not in percent.To get EVM Peak in percent, multiply by 100.
   - If MeasType = C0, the output is $20 \times \log 10\left(\frac{|C0|}{RMS(|S(k)|)}\right)$ dB
   - If MeasType = Frequency error, the output is $(Da)/(2 \times \pi \times SymTime)$ Hz
   - If MeasType = Droop, the output is $(-20 \times \log 10(e^{Dr}))$ dB
   - If MeasType = Error sequence, the output is the sequence $\frac{Re\{E(k)\} + j \times Im\{E(k)\}}{RMS(|S(k)|)}$
   - If MeasType = Magnitude Error sequence, the output is the magnitude error sequence (in Volts). For the definition of magnitude error, see the following figure.
   - If MeasType = Phase Error sequence, the output is the phase error sequence (in degrees). For the definition of phase error, see the following figure.
   - If MeasType = Sampled data, the output is the downsampled (1 sample per symbol period) input signal.

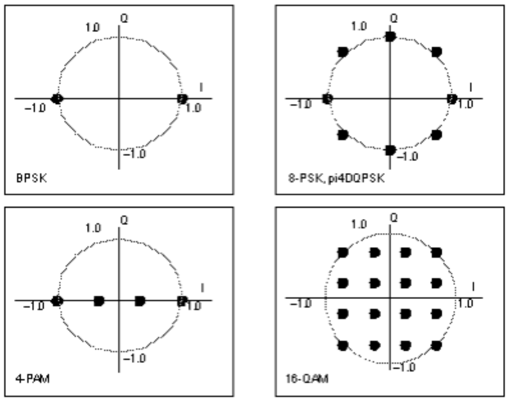


Magnitude and Phase Error

3. The ModType parameter is used to select one of the pre-defined or a user-defined signal constellation. All signal constellations (including user- defined) are normalized so that the maximum magnitude constellation point lies on the unit circle. For example, for a QPSK constellation, all four IQ points lie on the unit circle.
   For a PAM-type signal (includes BPSK, 4-PAM, 8-PAM) the Q-channel of the signal is set to 0.0.
   The following figure shows a few examples of the IQ constellation sets used in the EVM measurement.
   The HPSK option is for an HPSK signal that has equal gains applied to the I and Q channels resulting in a 4-point constellation. In effect, this is the same as a QPSK constellation. If the gains applied to the I and Q channels are different, the resulting constellation has 8 points and using the HPSK option gives the wrong results; in this

case, the *User defined* option should be used, which enables you to specify an arbitrary constellation in the Constellation parameter.

4. The Start and OptimizeSamplingInstant parameters define how the input signal is downsampled. Note that the EVM algorithm operates on the downsampled input signal.
   If OptimizeSamplingInstant is set to NO, the input signal is downsampled at one sample per symbol period starting at Start. If Start is not an exact multiple of the simulation time step, interpolation is used to find the input signal values in between the available samples.
   If OptimizeSamplingInstant is set to YES, the input signal is downsampled at one sample per symbol period starting at StartDownSampling, where StartDownSampling is swept from Start to (Start + SymTime) with a step of (simulation time step) / 10. This way the optimal sampling instant can be found. The optimal value for StartDownSampling is displayed on the status window and can be used in subsequent simulations as the value of Start with OptimizeSamplingInstant set to NO. This setting results in faster simulations and gives the correct results as long as no parameter that can affect the optimal sampling instant (for example the delay of a filter) is changed. If such a parameter changes, then OptimizeSamplingInstant should be set to YES.
   Another way to find when the optimal sampling instant is, without setting the OptimizeSamplingInstant parameter to YES, is to observe the eye diagram and find where its maximum opening occurs.

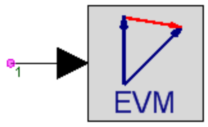


Examples of IQ-Constellations as Used in EVM Measurement

5. For general information regarding sinks, refer to *About Sinks* (algorithm).

See:
EVM_Env (algorithm)

EVM_Env



Description: Error Vector Magnitude Measurement
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *EVM Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Start	Start time for data collection	Start_Time	s	Float	NO	[0:∞)
SymTime	Symbol duration time	10e-6	s	Float	NO	[TStep:1000*TStep]†
SymBurstLen	Burst length in number of symbols	100		Integer	NO	[1:∞)
MeasType	Measurement type: EVM RMS, EVM Peak, C0, Frequency error, Droop, Error sequence, Magnitude error sequence, Phase error sequence, Sampled data	EVM RMS		Enumeration	NO	
ModType	Modulation type: BPSK, QPSK, HPSK, PI/4 DQPSK, PSK8, PSK16, QAM4, QAM16, QAM32, QAM64, QAM128, QAM256, PAM4, PAM8, User defined	QPSK		Enumeration	NO	
Constellation	Complex constellation values	[1+j, 1-j, -1-j, -1+j]		Complex array	NO	
OptimizeSamplingInstant	Automatically find optimal sampling instant: NO, YES	YES		Enumeration	NO	

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	envelope	NO

Notes/Equations

1. The EVM_Env model is used to perform an EVM (Error Vector Magnitude) measurement for an envelope signal. For more details on the EVM measurement see *EVM_Cx* (algorithm).
2. For general information regarding sinks, refer to *About Sinks* (algorithm).

See:
EVM_Cx (algorithm)

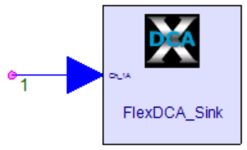
FlexDCA_Sink Part

Categories: *Sinks* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
FlexDCA_Sink (algorithm)	Stream simulation samples to the Agilent FlexDCA Application

FlexDCA_Sink



Description: Stream simulation samples to the Agilent FlexDCA Application
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *FlexDCA Sink Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable
NumberOfInputs	Number of inputs.	1		Integer	NO
ContinuousMode	Run simulation continuously.: NO, YES	YES		Enumeration	NO
NumBitsToSkip	Number of bits to skip before writing to text file.	0		Integer	NO
NumPatternToSimulate	Number of patterns to simulate.	1		Integer	NO
BitRate	Original bit rate.	Sample_Rate	Hz	Float	NO
PatternLength	Number of bits per period for periodic bit pattern.	127		Integer	NO
AdvancedParameters	Display advanced parameters.: NO, YES	NO		Enumeration	NO
SetupFile	Name of setup file to recall			Filename	NO

Simulation start always sets FlexDCA to Eye mode. The reason is that the amount of data buffered for jitter analysis is many magnitudes higher than the amount needed for Eye or Scope mode. By starting FlexDCA in Eye mode, old data can be quickly flushed out of the buffer. For the same reason, **if transient is observed in Jitter mode to make eye closed, switch to Eye mode until eye diagram is clear before switching back to Jitter mode.**

No more than one FlexDCA_Sink can be placed on the same schematic.

Before you start SystemVue, make sure **FlexDCA software is not already running**.

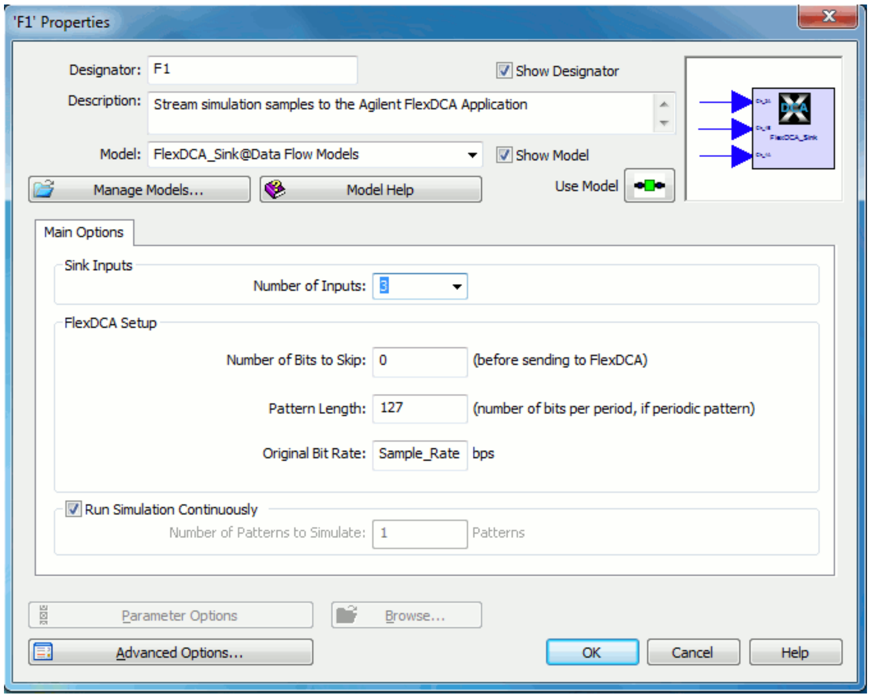
FlexDCA **requires** patterned (periodic) bits for jitter analysis. For example, use ***PRBS* (algorithm)** model in ***Bits Part* (algorithm)** for the bit source.

FlexDCA_Sink UI Properties

If desired, click the **Advanced Options ...** button on the bottom left corner of **FlexDCA_Sink UI** to view the parameter list. The mapping between the parameters in the parameter list and the setup controls shown in the Graphic User Interface described here should be straightforward.

Most of the setup controls in the User Interface is self explanatory. In the following we only cover a few key setup controls.

1. **Number of Inputs** FlexDCA can support up to 8 channels of inputs. Specify how many channels are desired and the input pins on the placed part will adjust accordingly.
2. **Number of Bits to Skip** Specify how many bits to skip before streaming the data to FlexDCA. Typically it is used to avoid sending transient data into the FlexDCA.
3. **Pattern Length** FlexDCA **requires** patterned (periodic) bits for jitter analysis, so provide pattern length here. For example, if you use ***PRBS* (algorithm)** model in ***Bits Part* (algorithm)** as the bit source and the LFSR_Length is 12, then the pattern length is 2^12 - 1 = 4095.
4. **Run Simulation Continuously** By default, simulation will run continuously and streaming bits into FlexDCA. If unchecked, please specify how many patterns to simulate.


'F1' Properties

Designator: F1 ☑ Show Designator

Description: Stream simulation samples to the Agilent FlexDCA Application

Model: FlexDCA_Sink@Data Flow Models ☑ Show Model

Manage Models... Model Help Use Model

Main Options

Sink Inputs

Number of Inputs: 3

FlexDCA Setup

Number of Bits to Skip: 0 (before sending to FlexDCA)

Pattern Length: 127 (number of bits per period, if periodic pattern)

Original Bit Rate: Sample_Rate bps

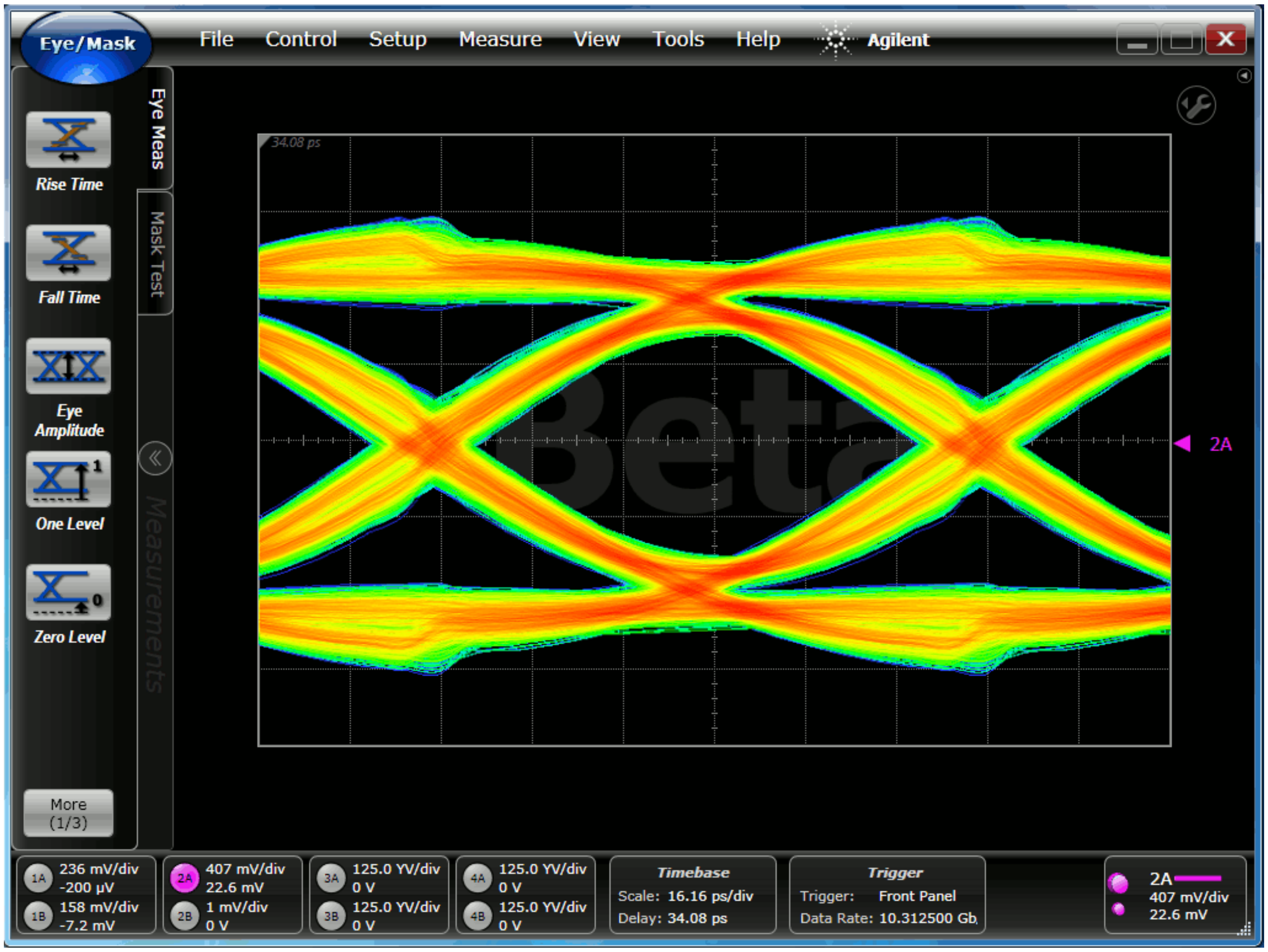
☑ Run Simulation Continuously

Number of Patterns to Simulate: 1 Patterns

Parameter Options Browse...

Advanced Options... OK Cancel Help

FlexDCA Control Basics



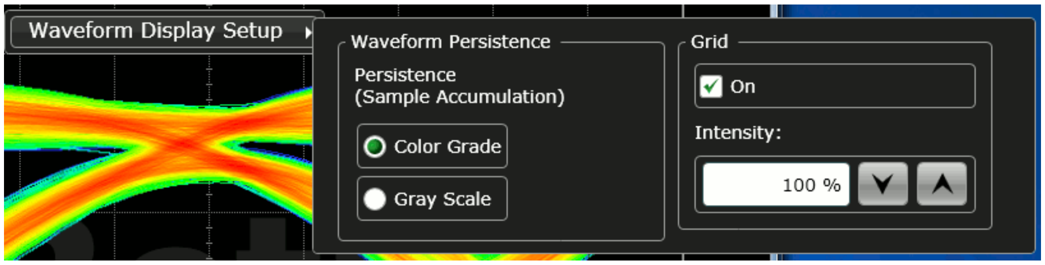
1A 236 mV/div -200 µV
1B 158 mV/div -7.2 mV
2A 407 mV/div 22.6 mV
2B 1 mV/div 0 V
3A 125.0 YV/div 0 V
3B 125.0 YV/div 0 V
4A 125.0 YV/div 0 V
4B 125.0 YV/div 0 V

Timebase
Scale: 16.16 ps/div
Delay: 34.08 ps

Trigger
Trigger: Front Panel
Data Rate: 10.312500 Gb

2A 407 mV/div 22.6 mV

- **Color Grade Display** To have the Eye diagram displayed in color grade, mouse right click on the eye diagram area, then select "**Color Grade**" as shown here:

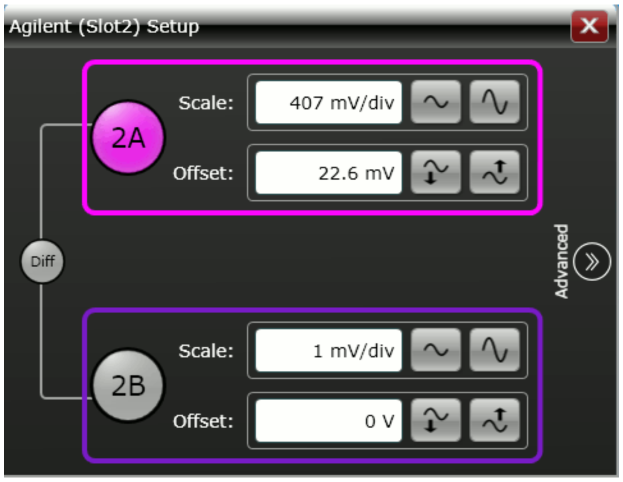


- **Mode Selection** Either use menu "**Setup -> Mode**" or the big ellipse-shaped button on the top-right corner of the display. The simulation will always set FlexDCA to its Eye mode.

⚠ **A separate license has to be purchased** in order to use the **Jitter** mode.

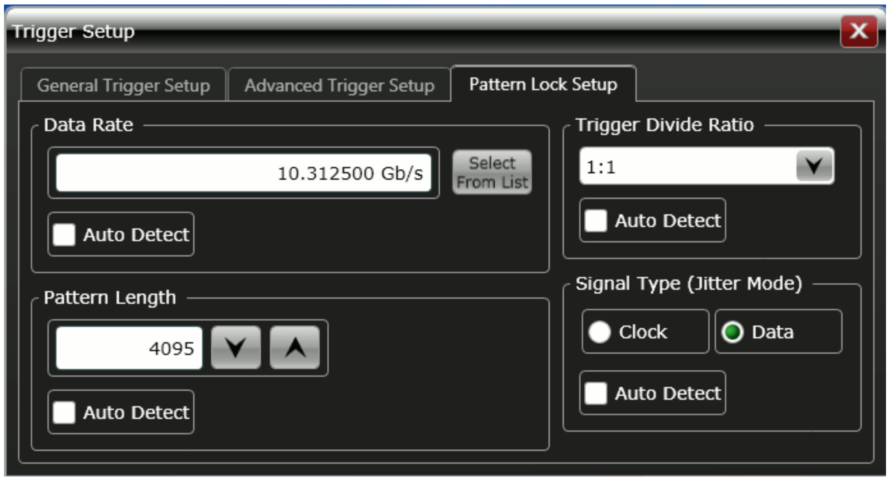
- **Channel Selection** Notice a row of rectangle areas on the bottom side of the display. Click on any one of the 4 rectangles on the right side will bring up a channel activation dialog that looks like the following. Toggle the round shaped button to turn ON/OFF the display for the channel.

⚠ In **Jitter** mode (which needs **a separate license**), only one channel can be active.



- **Auto-Scale** Use menu "**Control -> Auto Scale**" to auto-scale the displays as needed (e.g. if the Eye diagram is not centered).
- **Pattern Parameters** Critical pattern parameters such as Pattern Length and Data Rate are set up automatically at the start of simulation. However, you can modify them during simulation by clicking on the ***Trigger*** rectangle and go to the **Pattern Lock Setup** tab of the dialog opened as shown here.

⚠ At simulation start, these parameters will be reset by the simulation. Also, all the "**Auto Detect**" functions (and their corresponding check boxes) are disabled by the simulation.

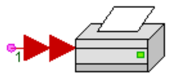


Notes/Links

1. An example of using FlexDCA_Sink can be found in **IBIS AMI Modeling.wsv** under ***Model Building Examples*** **(examples)**
2. To learn about FlexDCA and for software download, **go to**

http://www.agilent.com/find/FlexDCA

Printer (Data File Writer)



Description: Data File Writer
Category: Sinks
Domain: Timed
Associated Part:

Parameters

Name	Description	Default	Symbol	Unit	Type	Range
StartStopOption	Sink collection mode: Auto, Samples, Time	Auto			enum	
SampleStart	Sample number to start data collection	0			int	[0,∞)
SampleStop	Sample number to stop data collection	Num_Samples - 1			int	[SampleStart,∞)
TimeStart	Time to start data collection	Start_Time		sec	real	[0,∞)
TimeStop	Time to stop data collection	Stop_Time		sec	real	[TimeStart,∞)
File	Output filename	'print.txt'			filename	

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	multiple anytype	NO

Notes/Equations

1. Printer prints out one sample from each input port per line.
2. The output file is a text file that contains data in ADS Ptolemy format specific to the input data type: real array (for floating-point (real), fixed, and integer scalar input data), complex array, string array, real matrix, integer matrix, fixed-point matrix, or complex matrix. For format information, refer to Understanding File Formats in the *ADS Ptolemy Simulation* manual.
3. For general information regarding sinks, refer to *About Sinks* (algorithm).

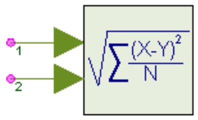
RMSE Part

Categories: *Sinks* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
RMSE (algorithm)	Root Mean Square Error Measurement

RMSE (Root Mean Square Error Measurement)



Description: Root Mean Square Error Measurement
Domain: Untimed
C++ Code Generation Support: NO
Associated Parts: *RMSE Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
StartFrame	Start frame	0		Integer	NO	[0:∞)
FramesToAverage	Number of frames to measure	1		Integer	NO	[1:∞)
FrameLength	Frame length in number of samples	Num_Samples		Integer	NO	[1:∞)
DisplayOption	Display option: RMS, dB	RMS		Enumeration	NO	

Input Ports

Port	Name	Description	Signal Type	Optional
1	InRef	Input reference signal	complex	NO
2	InTest	Input test signal	complex	NO

Notes/Equations

1. The RMSE model is used to calculate the root mean squared error between the two input signals.
2. Input data is collected starting at sample StartFrame × FrameLength and ending at sample (FramesToAverage + StartFrame) × FrameLength − 1.
3. The root mean square error is calculated according to the equation

$$RMSE = \frac{1}{N_f}\sum_{i=1}^{N_f}\sqrt{\frac{1}{L_f}\sum_{j}^{L_f}((I_1(i,j)-I_2(i,j))^2+(Q_1(i,j)-Q_2(i,j))^2)}$$

   where,
   N_f is the number of frames to average

   L_f is the frame length

   $I_1(i, j)$, Q_1 (i, j) and I_2 (i, j), Q_2 (i, j) are the in-phase and quadrature parts, respectively, of the input signals.
4. If DisplayOption is set to RMS, then the value as calculated by the above equations is saved in the dataset. If DisplayOption is set to dB, then 20×log10(RMSE) is saved in the dataset.

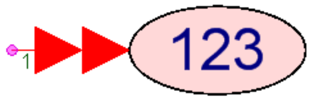
Sink Part

Categories: *Sinks* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
Sink (algorithm)	Data Sink

Sink (Data Sink)



Description: Data Sink
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Sink Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
StartStopOption	Sink collection mode: Auto, Samples, Time	Auto		Enumeration	NO	
SampleStart	Sample number to start data collection	0		Integer	NO	[0:∞)
SampleStop	Sample number to stop data collection	Num_Samples - 1		Integer	NO	[SampleStart:∞)
TimeStart	Time to start data collection	Start_Time	s	Float	NO	[0:∞)
TimeStop	Time to stop data collection	Stop_Time	s	Float	NO	[TimeStart:∞)
SinkTarget	Location where simulation data is written: DataSet, File, Both	DataSet		Enumeration	NO	
ContinuousMode	Run simulation in continuous mode: NO, YES	NO		Enumeration	NO	
WindowSize	Number of samples to collect and display in dynamically updated graph	500		Integer	NO	[1:∞)
DisableDataCollect	Disable the collection of data: NO, YES	NO		Enumeration	NO	
Graph	Graph to display the data in (if empty no graph is created)			Text	NO	
Table	Table to display the data in (if empty no table is created)			Text	NO	
BlockSize	Number of samples to collect from each input before writing to file as one data block	Num_Samples		Integer	NO	[1:∞)
ToSingleFile	Write data from all inputs to the same file: NO, YES	YES		Enumeration	NO	
DataFileName	Data file name	SinkData		Filename	NO	
DataFileType	Data file format: ASCII, Binary, SignalStudio, N5106A	ASCII		Enumeration	NO	
SkipFrequency	Do not write characterization frequency (for complex envelope data) to file: NO, YES	NO		Enumeration	NO	
NormalizeWaveform	Normalize input signal magnitude to [-1,1] range before writing to file: NO, YES	YES		Enumeration	NO	
ClippingLevel	Clipping voltage level	1.0	V	Float	NO	(0:∞)
Persistence	Keep sink data in dataset when workspace is saved: NO, YES	Data_Persistence		Enumeration	NO	

Input Ports

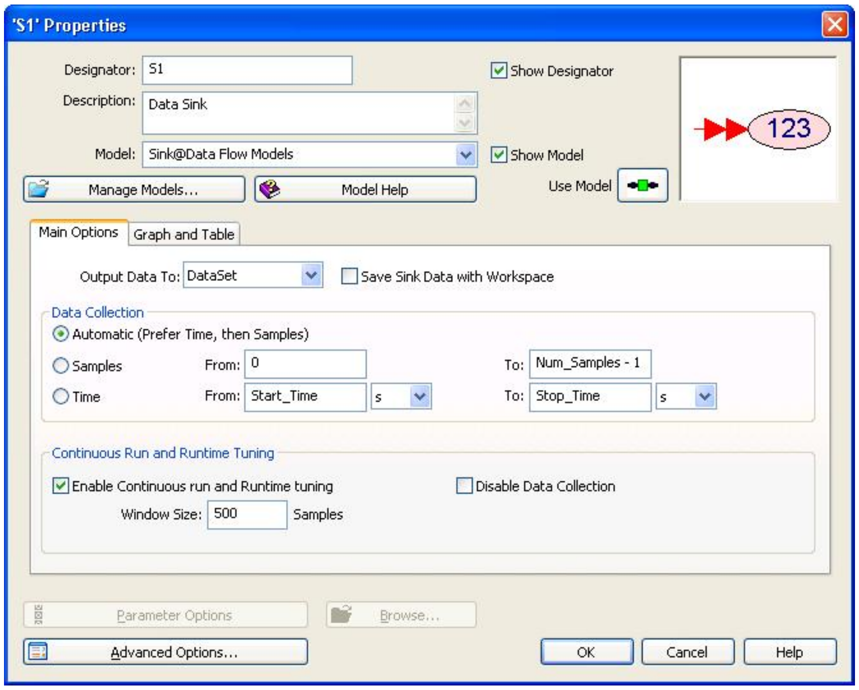
Port	Name	Description	Signal Type	Optional
1	input	Input Signal	multiple anytype	NO

Important Note: When writing data to files, **neither ASCII** file **nor Binary** file is for use with other Agilent products (such as VSA 89601A software). **ASCII** file is solely for logging simulation data when needed. **Binary** file is to temporarily store large amount of simulation data to be post processed by *MathLang script* (users). (See ***SignalDownloader_E4438C*** **(algorithm)** part as an example of how to use *MathLang script* (users) to post process **Binary** file).

Setup UI

A custom UI is provided for this model that allows easy set up.

Main Options Tab


'S1' Properties

Designator: S1
[✓] Show Designator
Description: Data Sink
Model: Sink@Data Flow Models
[✓] Show Model
Use Model

Main Options | Graph and Table

Output Data To: DataSet
[ ] Save Sink Data with Workspace

Data Collection
(●) Automatic (Prefer Time, then Samples)
( ) Samples From: 0 To: Num_Samples - 1
( ) Time From: Start_Time s To: Stop_Time s

Continuous Run and Runtime Tuning
[✓] Enable Continuous run and Runtime tuning
[ ] Disable Data Collection
Window Size: 500 Samples

The **Output Data To** drop down box provides three choices on how the simulation data will be stored:

- **DataSet**: simulation data is stored in a dataset; data in the dataset can be plotted in graphs/tables (see **Graph and Table Tab** on how to automatically create a graph or table at the end of the simulation) as well as post processed in equation pages.
- **File:** simulation data is written into one (or more) file(s).
- **Both:** simulation data is stored in a dataset as well as written into one (or more) file(s).
  When **File** or **Both** is selected a tab called **File Options Tab** appears that allows setting the file format as well as other properties of how data will be written to the file(s).

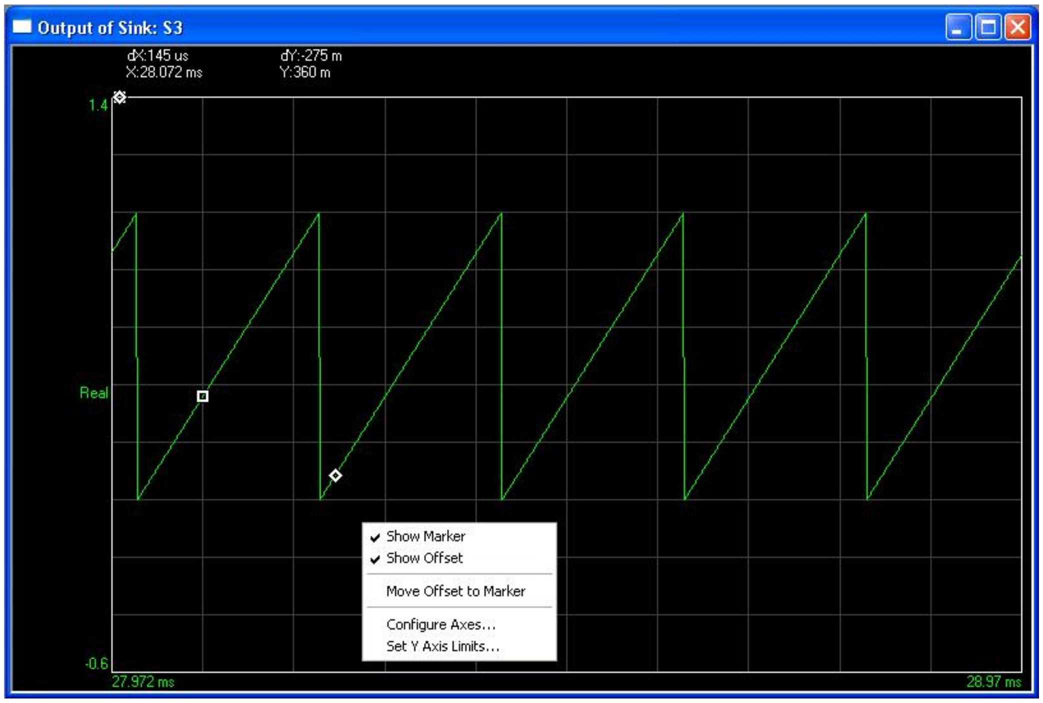
The **Save Sink Data with Workspace** checkbox allows you to save (checkbox is checked) or not save (checkbox is unchecked) the sink data with the workspace. Of course, this is only relevant when the simulation data is stored in a dataset (**DataSet** or **Both** selected in the **Output Data To** drop down box). By default, when the workspace is saved the data collected by the sink and stored in the dataset is not saved with the workspace. Not saving the simulation data collected by the sink with the workspace can significantly reduce the size of the workspace, which in turn speeds up the operations of saving and opening it. This is especially true for simulations that produce large amounts of data (millions of data points). For simulations that produce small amounts of data, not saving that data with the workspace is not going to provide any perceivable benefit. By default, the state of the **Save Sink Data with Workspace** checkbox is controlled by the *Data_Persistence* variable, which is tied to the state of the **Data Persistence** checkbox in the *Options tab of the Data Flow Analysis* (sim). When the **Data Persistence** checkbox is checked the value of the *Data_Persistence* variable is 1 and when the **Data Persistence** checkbox is unchecked the value of the *Data_Persistence* variable is 0.

The **Data Collection** area allows you to control how much simulation data will be collected. The amount of data to be collected can be specified in terms of **Samples** or **Time**. Start and Stop values for the data collection are specified in the **From** and **To** fields. Note that for data collection based on **Samples** sample indices start at 0. The variables *Num_Samples*, *Start_Time*, and *Stop_Time* used as default values in the **From** and **To** fields are defined in the **Number of Samples**, **Start Time**, and **Stop Time** fields in the *General tab of the Data Flow Analysis* (sim). The default data collection mode (**Automatic**) will do data collection based on **Time** if the input signal has a defined sample rate. Otherwise, it will do data collection based on **Samples**. The **Automatic** and **Samples** data collection modes are guaranteed to work for any simulation. The **Time** data collection mode will work only if the input signal has a defined sample rate. Otherwise, the simulation will terminate with an error.

The **Continuous Run and Runtime Tuning** area allows you to set up the simulation to run in an interactive continuous mode (when the **Enable Continuous run and Runtime tuning** checkbox is checked). In this mode:

- the simulation will keep running even after all the sinks have collected the data that has been requested
- for each sink that has the **Enable Continuous run and Runtime tuning** checkbox checked a dynamic plot is created where the input signal is plotted; since the dynamic plot can only plot a fixed number of samples (**Window Size** field) the waveform will appear to be moving (just as in the case of an oscilloscope or vector signal analyzer) unless it is periodic and the **Window Size** is set to a value that represents the exact number of samples in an integer number of periods
- all parameters/variables that are tunable can be tuned while the simulation is running
- the data collection (in the dataset or file) can be optionally disabled (**Disable Data Collection** checkbox)

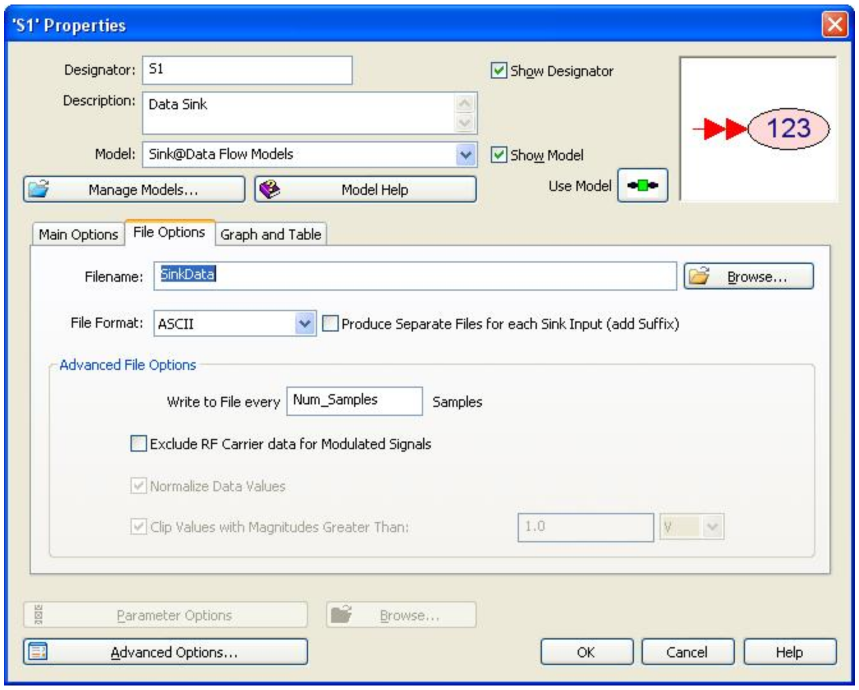
An example of a dynamic plot is shown below.



A right mouse click on the dynamic plot brings up a menu where you can

- enable a marker; its coordinates X and Y are shown at the top right corner of the plot
- enable an offset (marker); the differences between its coordinates and the coordinates of the marker are also shown at the top right corner of the plot (dX and dY)
- move the offset (marker) to the same point as the marker
- configure the axes; allows you to choose a number of different formats to be displayed: Log Mag (dB), Linear Mag, Real (I), Imag (Q), Wrap Phase, Unwrap Phase, I-Q, Constellation, Trelis-Eye, Group Delay, Log Mag (lin)
- set the Y-axis limits; allows you to select between auto-scaling the Y-axis or fixing its top and bottom limits

File Options Tab


'S1' Properties

Designator: S1 — [✓] Show Designator

Description: Data Sink

Model: Sink@Data Flow Models — [✓] Show Model

Use Model

Main Options | File Options | Graph and Table

Filename: SinkData

File Format: ASCII — [ ] Produce Separate Files for each Sink Input (add Suffix)

Advanced File Options

Write to File every Num_Samples Samples

[ ] Exclude RF Carrier data for Modulated Signals

[✓] Normalize Data Values

[✓] Clip Values with Magnitudes Greater Than: 1.0 V

The **Filename** field sets the name of the file where the simulation data will be written. If a relative path is specified, then the path is relative to the directory where the workspace is located. For a new workspace that has not been saved yet, the path is relative to the workspace that was the starting point. For example, if you create a new workspace starting with the Blank template then the paths are relative to the Templates directory of your SystemVue installation.

The **Produce Separate Files for each Sink Input (add Suffix)** checkbox allows (when checked) the creation of a separate file for each input signal. The first input signal uses the filename that was entered in the **Filename** field. The other inputs use filenames with the suffixes _1, _2, ... For example, if the filename entered is LTE.wfm, the second input will use filename LTE_1.wfm and so on. If this checkbox is not checked then all the input signals are saved in the same file.

The **File Format** drop down box is used to select the format of the file to be written. The supported formats are:

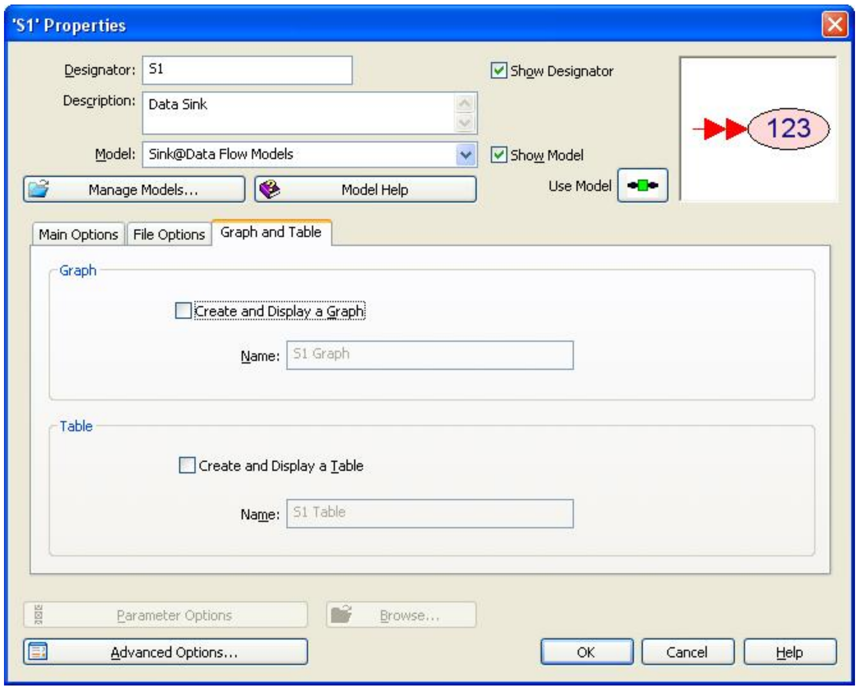
- **ASCII**: This creates an ASCII text file with the values of the input signals. Each input is written in its own column (if the **Produce Separate Files for each Sink Input (add Suffix)** checkbox is not checked). In addition, if the input signal has a defined sample rate, then the first column has the time stamps of the signal samples.
- **Binary**: In this format, every sample of the input signal is written to the file using 8 bytes **"double"** format. If the input signal is complex, the real and imaginary parts are written in the order of: #1 Real, #1 Imaginary, #2 Real, #2 Imaginary, #3 Real, #3 Imaginary, ... When there are multiple input signals written into a **single** file (**Produce Separate Files for each Sink Input (add Suffix)** checkbox is not checked), the order of the data is also impacted by the value of the **Write to File every ___ Samples** field. Let's assume the sink has been set up to **Write to File every 10 Samples**, and there are **3** complex input signals (s1, s2, s3). The order of the data in the file will be (the lines shown below separating the data points are only for readability; binary files do not have a concept of lines):
  s1 #1 Real, s1 #1 Imaginary, s1 #2 Real, s1 #2 Imaginary, ..., s1 #10 Real, s1 #10 Imaginary,
  s2 #1 Real, s2 #1 Imaginary, s2 #2 Real, s2 #2 Imaginary, ..., s2 #10 Real, s2 #10 Imaginary,
  s3 #1 Real, s3 #1 Imaginary, s3 #2 Real, s3 #2 Imaginary, ..., s3 #10 Real, s3 #10 Imaginary,
  s1 #11 Real, s1 #11 Imaginary, s1 #12 Real, s1 #12 Imaginary, ..., s1 #20 Real, s1 #20 Imaginary,
  s2 #11 Real, s2 #11 Imaginary, s2 #12 Real, s2 #12 Imaginary, ..., s2 #20 Real, s2 #20 Imaginary,
  s3 #11 Real, s3 #11 Imaginary, s3 #12 Real, s3 #12 Imaginary, ..., s3 #20 Real, s3 #20 Imaginary,
  s1 #21 Real, s1 #21 Imaginary, s1 #22 Real, s1 #22 Imaginary, ..., s1 #30 Real, s1 #30 Imaginary,
  s2 #21 Real, s2 #21 Imaginary, s2 #22 Real, s2 #22 Imaginary, ..., s2 #30 Real, s2 #30 Imaginary,
  s3 #21 Real, s3 #21 Imaginary, s3 #22 Real, s3 #22 Imaginary, ..., s3 #30 Real, s3 #30 Imaginary,

...
...
- **SignalStudio**: This is the Agilent Signal Studio encrypted waveform format.
- **N5106A**: This is a binary waveform format that is targeted for use by the Agilent N5106A PXB MIMO Receiver Tester .

The **Advanced File Options** area allows control of advanced properties for file writing:

- The **Write to File every ___ Samples** field sets the block size in which the simulation data is written to the file(s). For example, if the sink has been set up to collect 1000 samples and **Write to File every ___ Samples** is set to 100, then the sink will write data to the file each time it collects 100 samples (a total of 10 times). In the case of multiple input signals the value of this field impacts the order data is written in the **Binary** format as described above. Setting this field to a large value means the simulator has to store large amounts of data in memory before writing it to the file. Setting this field to a small value means the simulator has to access the file many times, which may slow down the simulation. A value of approximately 1M samples provides a good trade off between memory consumption and speed.
- The **Exclude RF Carrier data for Modulated Signals** checkbox can be used to disable (when checked) writing the input signal characterization frequency to the file. This is only relevant when the **ASCII** or **Binary** formats are used. If left unchecked, the input signal characterization frequency will be written to the file. In the case of **ASCII** format, the characterization frequency is written in its own column before the columns representing the signal values but after the column representing the time stamps (when the input signal has a defined sample rate). In the case of **Binary** format, a block of **Write to File every ___ Samples** values equal to the input signal characterization frequency is written before all blocks of the same size holding input signal values.
- The **Normalize Data Values** checkbox allows normalization of the input signal values before writing them to the file. This is only relevant when **SignalStudio** or **N5106A** formats are used. When this checkbox is checked, the input signal values are normalized so that both the real and imaginary parts fall in the range [-1.0, 1.0]. This is done by dividing all input values with the maximum absolute real or imaginary part value, that is, max(abs(Re{input[i]}), abs(Im{input[i]})), where input[i] is the i^{th} collected input sample and the maximum is over all collected samples. When this checkbox is left unchecked the input signal values are written to the file without any pre or post processing.
- The **Clip Values with Magnitudes Greater Than** checkbox allows limiting the magnitude of the real and imaginary part values that are written to the file (when the checkbox is checked) to a desired level (set in the field to the right of the checkbox). This is only relevant when **SignalStudio** or **N5106A** formats are used and the **Normalize Data Values** checkbox is not checked. Limiting the magnitude of the real and imaginary part values is useful in the cases where these files are intended to be played back in an arbitrary waveform generator instrument. The reason is that these instruments have a predefined full scale level and will clip anything above it (for Agilent instruments this level is typically 1V). When this checkbox is checked and clipping does occur a warning message is issued. This way a user can decide whether they want to scale their data to avoid clipping from occurring when the files are played back in the instrument. When this checkbox is left unchecked the input signal values are written to the file without any pre or post processing.

Graph and Table Tab


'S1' Properties

Designator: S1 — [✓] Show Designator

Description: Data Sink

Model: Sink@Data Flow Models — [✓] Show Model

Use Model

Graph and Table

Graph

[ ] Create and Display a Graph

Name: S1 Graph

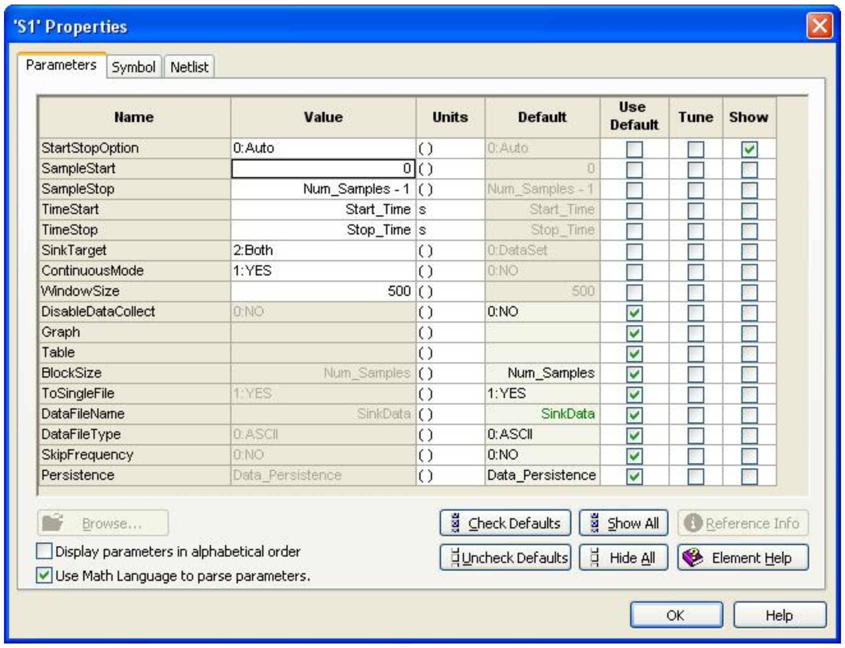
Table

[ ] Create and Display a Table

Name: S1 Table

This tab allows you to automatically create a graph and/or table displaying the collected data at the end of the simulation. Simply check the appropriate checkboxes in the **Graph** and/or **Table** areas. The name of the graph/table can also be specified. By default these names are the same as the sink's instance name followed by the word Graph or Table. This tab is available only when the simulation data is stored in a dataset (**DataSet** or **Both** selected in the **Output Data To** drop down box).

Advanced Options


'S1' Properties

Parameters

Name	Value	Units	Default	Use Default	Tune	Show
StartStopOption	0:Auto	( )	0:Auto	☐	☐	✓
SampleStart	0	( )	0	☐	☐	☐
SampleStop	Num_Samples - 1	( )	Num_Samples - 1	☐	☐	☐
TimeStart	Start_Time	s	Start_Time	☐	☐	☐
TimeStop	Stop_Time	s	Stop_Time	☐	☐	☐
SinkTarget	2:Both	( )	0:DataSet	☐	☐	☐
ContinuousMode	1:YES	( )	0:NO	☐	☐	☐
WindowSize	500	( )	500	☐	☐	☐
DisableDataCollect	0:NO	( )	0:NO	✓	☐	☐
Graph		( )		✓	☐	☐
Table		( )		✓	☐	☐
BlockSize	Num_Samples	( )	Num_Samples	✓	☐	☐
ToSingleFile	1:YES	( )	1:YES	✓	☐	☐
DataFileName	SinkData	( )	SinkData	✓	☐	☐
DataFileType	0:ASCII	( )	0:ASCII	✓	☐	☐
SkipFrequency	0:NO	( )	0:NO	✓	☐	☐
Persistence	Data_Persistence	( )	Data_Persistence	✓	☐	☐

[ ] Display parameters in alphabetical order

[✓] Use Math Language to parse parameters.

Pressing the **Advanced Options** button brings up a dialog box with the sink's parameters in table form. This allows setting the values of parameters that are represented with drop down boxes, radio buttons, or checkboxes in the custom UI to variables. The following table describes the association of these parameters to the different UI controls.

Parameter Name	**Associated UI control**	**Tab UI control is located**
StartStopOption	**Automatic/Samples/Time** radio buttons	Main Options
SampleStart	**From** field next to **Samples** radio button	Main Options
SampleStop	**To** field next to **Samples** radio button	Main Options
TimeStart	**From** field next to **Time** radio button	Main Options
TimeStop	**To** field next to **Time** radio button	Main Options
SinkTarget	**Output Data To** drop down button	Main Options
ContinuousMode	**Enable Continuous run and Runtime tuning** checkbox	Main Options
WindowSize	**Window Size** field	Main Options
DisableDataCollect	**Disable Data Collection** checkbox	Main Options
Graph	**Create and Display a Graph** checkbox and associated **Name** field	Graph and Table
Table	**Create and Display a Table** checkbox and associated **Name** field	Graph and Table
BlockSize	**Write to File every ____ Samples** field	File Options
ToSingleFile	**Produce Separate Files for each Sink Input (add Suffix)** checkbox	File Options
DataFileName	**Filename** field	File Options
DataFileType	**File Format** drop down box	File Options
SkipFrequency	**Exclude RF Carrier data for Modulated Signals** checkbox	File Options
NormalizeWaveform	**Normalize Data Values** checkbox	File Options
ClippingLevel	**Clip Values with Magnitudes Greater Than** checkbox and associated field	File Options
Persistence	**Save Sink Data with Workspace** checkbox	Main Options

MathLang Equation in Sink

You can execute a *MathLang Equation* (users) in Sink to post-process simulation data, although the preferred method is to create an Equation object on the workspace tree and doing any post-processing there.

Where the Sink's built-in MathLang equation execution becomes handy is when you need to perform an action when a Sink is done collecting data, such as downloading data to an instrument. Two good examples are the E4438C_SignalDownloader part and the N5106A_SignalDownloader part. These two parts download simulation waveforms into instruments and are sub-circuit parts that build on the Sink part. They both use the Math Language code in the Sink Equation to communicate and control the instruments for instruments setup and waveform downloading.

⚠ Currently, the Equation provided to the Sink will only be executed at the end of the simulation. It does ***not*** recognize the functions such as Initialize(), Run() and Finalize() that are supported by the *MathLang* (algorithm) part. Support for these functions in the Sink will be available in future releases of SystemVue.

ⓘ To create or access the equation used by the Sink, simply right click on the part after it is placed inside the schematic of your design, and choose "Edit Equation". An equation editor will appear.

M_State

You can access simulation information from a Math Language variable built into the Sink called **M_State**. The **M_State** variable is a structure that is pre-filled with some fields described below. For example, you can access the sampling rate for the simulation data using the following syntax in the Equation:

```
sampleRate = M_State.SampleRate;
```

The following describes the fields provided to you in the **M_State** structure:

Fc	Carrier frequency for the simulation data if they are for modulated RF signal
FileName	When written data into a file, this is the absolute file name, i.e. it includes absolute path to the file
MaxVal	The maximum value of all the samples captured by the Sink. In complex samples, it is the maximum of all the real and imaginaryvalues.
MinVal	Opposite to **MaxVal**
NumberOfInputs	The number of input streams that go into the Sink
NumberOfSamples	The number of samples from **each individual input stream** that are captured by the Sink
SampleRate	Sample rate of the simulation data

For examples of how to use the information from **M_State**, see the **SignalDownloader_E4438C** part and the **SignalDownloader_N5106A** part. Both are parts that use sub-network models that embed a Sink in their sub-network. The Equation for the Sink post processes the simulation data before sending it to the instruments, and it relies on **M_State** to provide critical instrument set up information such as carrier frequency, sampling rate, waveform file name, etc.

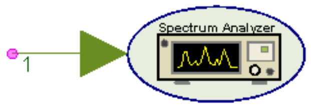
SpectrumAnalyzer Part

Categories: *Sinks* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SpectrumAnalyzerEnv (algorithm)	Spectrum Analyzer for Complex Envelope Signals
SpectrumAnalyzerCx (algorithm)	Spectrum Analyzer for Real and Complex Signals

SpectrumAnalyzerCx (SpectrumAnalyzerCx)



Description: Spectrum Analyzer for Real and Complex Signals
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *SpectrumAnalyzer Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Mode	Mode of operation: TimeGate, ResBW	TimeGate		Enumeration	NO	
Start	Start time for data collection	Start_Time	s	Float	NO	[0:∞)
SegmentTime	Segment time	Stop_Time - Start_Time + Time_Spacing	s	Float	NO	[16*TStep:∞)†
ResBW	Resolution bandwidth	Freq_Resolution	Hz	Float	NO	(0:∞)
NumSegments	Number of segments to be processed	1		Integer	NO	[1:∞)
Overlap	Segment overlap in percent	0.0		Float	NO	[0:100)
Window	Window applied to collected data: Uniform, Hanning, Gaussian Top, Flat Top, Blackman Harris	Uniform		Enumeration	NO	
FStart	Start frequency	-100.0e9	Hz	Float	NO	(-∞:∞)
FStop	Stop frequency	100.0e9	Hz	Float	NO	(FStart:∞)
SpectrumType	Output spectrum type: Complex Voltage, Power/Phase	Power/Phase		Enumeration	NO	
SpectrumDisplay	Spectrum display option: Double Sided, Single Sided	Double Sided		Enumeration	NO	
RefR	Reference resistance	50.0	ohm	Float	NO	(0:∞)
Persistence	Not saving sink data in the workspace file reduces the file size: NO, YES	Data_Persistence		Enumeration	NO	

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	complex	NO

⚠ In releases prior to 2009.05, in order to get a clean spectrum for a periodic signal, the *SegmentTime* parameter had to be set to $T - TStep$, where T is the signal period and *TStep* is the simulation time step for the input signal.
In release 2009.05 (and later), in order to get a clean spectrum for a periodic signal, the *SegmentTime* parameter has to be set to T.
As a result, workspaces created with the 2008.12 release might not give identical results when simulated in the 2009.05 (or later) release. This is true regardless of whether the input signal is periodic or not.

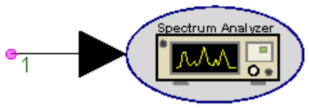
Notes/Equations

1. The SpectrumAnalyzerCx model can be used to measure the spectrum of a real or a complex signal. It works very similar to the *SpectrumAnalyzerEnv* (algorithm) model. The only difference is that it can display *Single Sided* or *Double Sided* spectra (set in the *SpectrumDisplay* parameter). *Single Sided* spectra are valid only when the input signal is real. If the input signal is complex then the value of the *SpectrumDisplay* parameter is ignored and the spectrum computed and displayed is always the *Double Sided* one.

- In a *Double_Sided* spectrum, each spectral tone represents an $A \cdot e^{jw_c t}$ term, where *A* is a complex number. Since $e^{jw_c t}$ has constant magnitude of 1, the associated power is $A^2/RefR$.
  When *SpectrumType* is *Complex Voltage* the resulting spectrum is the complex values *A* at each frequency.
  When *SpectrumType* is *Power/Phase* the resulting spectra are the power ($A^2/RefR$) and phase (phase of *A*) at each frequency.
- In a *Single_Sided* spectrum, each spectral tone represents an $A \cdot cos(w_c t + \theta)$ term, where *A* and θ are real numbers. The associated power is $A^2/(2 \cdot RefR)$, since for a non-constant magnitude signal we need to use its rms value (for power calculations).
  When *SpectrumType* is *Complex Voltage* the resulting spectrum is the complex values $A \cdot (cos(\theta) + j \cdot sin(\theta))$ at each frequency.
  When *SpectrumType* is *Power/Phase* the resulting spectra are the power ($A^2/(2 \cdot RefR)$) and phase (θ) at each frequency.

2. This model can be used directly with complex signals, whereas in order to connect a complex signal to the *SpectrumAnalyzerEnv* (algorithm) a *CxToEnv* (algorithm) converter has to be placed in between.
3. For more details see the documentation of *SpectrumAnalyzerEnv* (algorithm).

SpectrumAnalyzerEnv (SpectrumAnalyzerEnv)



Description: Spectrum Analyzer for Complex Envelope Signals
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *SpectrumAnalyzer Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
Mode	Mode of operation: TimeGate, ResBW	TimeGate		Enumeration	NO	
Start	Start time for data collection	Start_Time	s	Float	NO	[0:∞)
SegmentTime	Segment time	Stop_Time - Start_Time + Time_Spacing	s	Float	NO	[16*TStep:∞)†
ResBW	Resolution bandwidth	Freq_Resolution	Hz	Float	NO	(0:∞)
NumSegments	Number of segments to be processed	1		Integer	NO	[1:∞)
Overlap	Segment overlap in percent	0.0		Float	NO	[0:100)
Window	Window applied to collected data: Uniform, Hanning, Gaussian Top, Flat Top, Blackman Harris	Uniform		Enumeration	NO	
FStart	Start frequency	0.0	Hz	Float	NO	[0:∞)
FStop	Stop frequency	100.0e9	Hz	Float	NO	(FStart:∞)
SpectrumType	Output spectrum type: Complex Voltage, Power/Phase	Power/Phase		Enumeration	NO	
RefR	Reference resistance	50.0	ohm	Float	NO	(0:∞)
Persistence	Not saving sink data in the workspace file reduces the file size: NO, YES	Data_Persistence		Enumeration	NO	

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	input signal	envelope	NO

⚠ In releases prior to 2009.05, in order to get a clean spectrum for a periodic signal, the *SegmentTime* parameter had to be set to *T* − *TStep*, where *T* is the signal period and *TStep* is the simulation time step for the input signal.
In release 2009.05 (and later), in order to get a clean spectrum for a periodic signal, the *SegmentTime* parameter has to be set to *T*.
As a result, workspaces created with the 2008.12 release might not give identical results when simulated in the 2009.05 (or later) release. This is true regardless of whether the input signal is periodic or not.

Notes/Equations

1. The SpectrumAnalyzer model can be used to measure the spectrum of a real

baseband or a complex envelope signal. In the following notes TStep is used to denote the simulation time step and fc is used to denote the signal characterization frequency (fc = 0 for a real baseband signal and fc>0 for a complex envelope signal).

2. The SpectrumAnalyzer has two modes of operation: *TimeGate* and *ResBW*. The mode of operation is selected in the *Mode* parameter.
   In both modes of operation the model collects data starting at *Start* and until *NumSegments* segments that overlap by *Overlap%* are collected. Each segment is windowed by the specified *Window*, the spectrum of each segment is computed, and finally the spectra are averaged. The only difference between the two modes of operation is that
   - in *TimeGate* mode, the duration of each segment is *SegmentTime*.
   - in *ResBW* mode, the duration of each segment is *NENBW/ResBW*, where *NENBW* is the the Normalized Equivalent Noise Bandwidth of the window used.
3. The spectrum of a basedband signal extends from 0 Hz to 1/(2 × *TStep*) Hz. The spectrum of an RF signal extends from *fc* − 1/(2 × *TStep*) Hz to *fc* + 1/(2 × *TStep*) Hz. When *fc* < 1/(2 × *TStep*), the spectrum extends to negative frequencies. The spectral content at these negative frequencies is conjugated, mirrored, and added to the spectral content of the closest positive frequency. This way, the negative frequency tones are displayed on the positive frequency axis as would happen in a real spectrum analyzer measurement instrument. This process can introduce an error in the displayed frequency for the mirrored tones. The absolute error introduced is less than *Δf/2*, where *Δf* is *1/SegmentTime* (in *TimeGate* mode) and *ResBW/NENBW* (in *ResBW* mode). The *FStart* and *FStop* parameters are used to control the frequencey range over which the spectrum will be calculated. If *FStart* and *Fstop* are outside the limits mentioned above they are reset to the lower and higher limits respectively.
4. The *SpectrumType* parameter is used to select the type of spectrum to be computed. The computed spectrum is always a *Single Sided* spectrum. See *SpectrumAnalyzerCx* (algorithm) for details regarding *Single Sided* and *Double Sided* spectra and the interpretation of *Complex Voltage* and *Power/Phase* spectra.
   - If *SpectrumType* is *Complex Voltage*, then the computed spectrum is the complex amplitude voltage spectrum. The result is saved in the dataset in a complex variable named <InstanceName>, where <InstanceName> is the model's instance name.
   - If *SpectrumType* is *Power/Phase*, then there are two spectra computed: the power spectrum and the phase spectrum. The results are saved in the dataset in two real variables named <InstanceName>_Power and <InstanceName>_Phase, where <InstanceName> is the model's instance name.
5. **Windowing in spectral analysis and how to choose the right window**

   Windowing is necessary in transform-based (FFT) spectrum estimation. Without windowing, the estimated spectrum can suffer from spectral leakage that can cause misleading measurements or masking of weak signal spectral detail by spurious artifacts.
   Every time a window is applied to a signal, leakage occurs, that is, power from one spectral component leaks into the adjacent ones. Leakage from strong spectral components can result in hiding/masking of nearby weaker spectral components. Even strong spectral components can be affected by leakage. For example, two strong spectral components close to each other can appear as one due to leakage.
   Choosing the right window for a spectral measurement is very important. The choice of window depends on what is being measured and what the trade-offs between frequency resolution (ability to distinguish spectral components of comparable strength that are close to each other) and dynamic range (ability to measure signals with spectral components of widely varying strengths and distributed over a wide range) are.
   Windows can be characterized by their Normalized Equivalent Nosie BandWidth (NENBW). Equivalent Noise Bandwidth (ENBW) compares a window to an ideal, rectangular filter. It is the equivalent width of a rectangular filter that passes the same amount of white noise as the window. The Normalized ENBW (NENBW) is the ENBW multiplied by the time duration of the signal being windowed. In general, for the same length of signal processed, the higher the NENBW of a window the higher its dynamic range (less leakage) and the poorer its frequency resolution. The NENBW of the windows available in the SpectrumAnalyzer is given in the table below:

Window	NENBW
Uniform	1
Hanning	1.5
Gaussian Top	2.215
Flat Top	3.819
Blackman Harris	2.021

   Some general guidelines for choosing a window are given below:

   - Do not use a window (set *Window* to *Uniform*) when analyzing transients.
   - For periodic signals whose spectral components have comparable strengths and when the signal segment processed includes an exact integer multiple of periods, the best results are obtained if no window is used (set *Window* to *Uniform*). Any start up transients should be excluded.

- For periodic signals whose spectral components have significantly different strengths and/or when the signal segment processed does not include an exact integer multiple of periods, the use of a window can improve the detection of the weaker spectral components. The higher the NENBW the more likely the weaker spectral components will be detected. However, this trades-off frequency resolution and so if the spectral components are very close to each other the weaker one might remain unresolved. To improve frequency resolution while still maintaining a good dynamic range use a window but process a longer signal segment.
- For aperiodic signals such as modulated signals (QPSK, QAM, GSM, EDGE, CDMA, OFDM) the use of a window is highly recommended. The window will attenuate the signal at both ends of the signal segment processed to zero. This makes the signal apear periodic and reduces leakage.

6. The definitions of the windows available in the SpectrumAnalyzer model are given below (*N* is the length of the window in number of samples):
   - $\text{Uniform } w(k) = \begin{cases} 1.0 & \text{if } 0 \le k \le N \\ 0.0 & \text{otherwise} \end{cases}$
   - $\text{Hanning } w(k) = \begin{cases} 0.5 - 0.5 \cdot \cos(\frac{2\pi k}{N}) & \text{if } 0 \le k \le N \\ 0.0 & \text{otherwise} \end{cases}$
   - $\text{Gaussian Top } w(k) = \begin{cases} 0.323 - 0.471 \cdot \cos(\frac{2\pi k}{N}) + 0.176 \cdot \cos(\frac{4\pi k}{N}) - 0.0285 \cdot \cos(\frac{6\pi k}{N}) + 0.00126 \cdot \cos(\frac{8\pi k}{N}) & \text{if } 0 \le k \le N \\ 0.0 & \text{otherwise} \end{cases}$
   - $\text{Flat Top } w(k) = \begin{cases} 0.213 - 0.414 \cdot \cos(\frac{2\pi k}{N}) + 0.279 \cdot \cos(\frac{4\pi k}{N}) - 0.0862 \cdot \cos(\frac{6\pi k}{N}) + 0.00749 \cdot \cos(\frac{8\pi k}{N}) & \text{if } 0 \le k \le N \\ 0.0 & \text{otherwise} \end{cases}$
   - $\text{Blackman Harris } w(k) = \begin{cases} 0.355768 - 0.487396 \cdot \cos(\frac{2\pi k}{N}) + 0.144232 \cdot \cos(\frac{4\pi k}{N}) - 0.012604 \cdot \cos(\frac{6\pi k}{N}) & \text{if } 0 \le k \le N \\ 0.0 & \text{otherwise} \end{cases}$

References

1. A. V. Oppenheim, R. W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, 1989, Chapter 9, section 9.7.

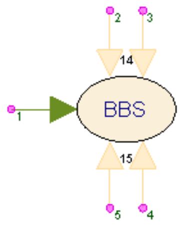
WriteBaseBandStudioFile Part

Categories: *Sinks* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
WriteBaseBandStudioFile (algorithm)	Base Band Studio Formatted File Writer
WriteBaseBandStudioFileNormalize (algorithm)	Base Band Studio Formatted File Writer with Input Normalization

WriteBaseBandStudioFile



Description: Base Band Studio Formatted File Writer
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *WriteBaseBandStudioFile Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
File	Input file name	file.bin		Filename	NO		F
StartStopOption	Sink collection mode: Auto, Samples, Time	Auto		Enumeration	NO		
SampleStart	Sample number to start data collection	0		Integer	NO	[0:∞)	
SampleStop	Sample number to stop data collection	Num_Samples - 1		Integer	NO	[SampleStart:∞)	
TimeStart	Time to start data collection	Start_Time	s	Float	NO	[0:∞)	
TimeStop	Time to stop data collection	Stop_Time	s	Float	NO	[TimeStart:∞)	

Input Ports

Port	Name	Description	Signal Type	Optional
1	IQ	IQ data	complex	NO
2	Marker14R	Real Bit 14 Marker	int	YES
3	Marker14I	Imag Bit 14 Marker	int	YES
4	Marker15R	Real Bit 15 Marker	int	YES
5	Marker15I	Imag Bit 15 Marker	int	YES

Notes/Equations

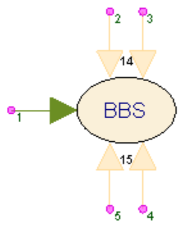
1. Write IQ complex data and boolean markers in the Base Band Studio format to a file named by the File parameter.
2. Each Base Band Studio sample consists of a pair of 16 bit signed integers which encode the IQ and up to 4 marker values. The first integer hold the real parts while the second integer hold the imaginary parts.
3. If Marker14R or Marker14I is connected, the IQ will have 14 bit quantization. If only Marker15R or Marker15I is connected, then the IQ will have 15 bit quantization. Otherwise, the IQ will have 16 bit quantization.
4. The IQ input is a complex value with the I value in the imaginary part and the Q value in the real part. If the input is outside the range [-1, 1), the value is clipped to this range before quantization.
5. A marker input is resolved to a zero bit, only if the marker input is zero. The marker bits occupy the least significant bits (LSB) of both integers. The marker name, e.g. Marker15R, indicate which bit and which integer hold the bit value, e.g. Marker15R states the LSB (15) of the first integer (R) is used store Marker15R. The next greater LSB is bit 14.
6. A Browse button is provided at the lower left corner of the properties window to provide a relative path for File. This path is relative to the folder from which the workspace is loaded. If an absolute path is desired, directly enter the full path into the Value column.

7. See *Sink* (algorithm) for a description of the output control parameters.
8. The Base Band Studio format has implementation in ADS and is used in Agilent instrumentation. For more information, see CM BB StudioStreamWrite .

Important Links

1. Learn more about Baseband Studio from Agilent Technologies
2. Also see the following related parts:
   ReadBaseBandStudioFile (algorithm)
   WriteBaseBandStudioFileNormalize (algorithm)

WriteBaseBandStudioFileNormalize



Description: Base Band Studio Formatted File Writer
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *WriteBaseBandStudioFile Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
File	Input file name	file.bin		Filename	NO		F
StartStopOption	Sink collection mode: Auto, Samples, Time	Auto		Enumeration	NO		
SampleStart	Sample number to start data collection	0		Integer	NO	[0:∞)	
SampleStop	Sample number to stop data collection	Num_Samples - 1		Integer	NO	[SampleStart:∞)	
TimeStart	Time to start data collection	Start_Time	s	Float	NO	[0:∞)	
TimeStop	Time to stop data collection	Stop_Time	s	Float	NO	[TimeStart:∞)	

Input Ports

Port	Name	Description	Signal Type	Optional
1	IQ	IQ data	complex	NO
2	Marker14R	Real Bit 14 Marker	int	YES
3	Marker14I	Imag Bit 14 Marker	int	YES
4	Marker15R	Real Bit 15 Marker	int	YES
5	Marker15I	Imag Bit 15 Marker	int	YES

Notes/Equations

1. Write IQ complex data and boolean markers in the Base Band Studio format to a file named by the File parameter.
2. Each Base Band Studio sample consists of a pair of 16 bit signed integers which encode the IQ and up to 4 marker values. The first integer hold the real parts while the second integer hold the imaginary parts.
3. If Marker14R or Marker14I is connected, the IQ will have 14 bit quantization. If only Marker15R or Marker15I is connected, then the IQ will have 15 bit quantization. Otherwise, the IQ will have 16 bit quantization.
4. The IQ input is a complex value with the I value in the imaginary part and the Q value in the real part. The I and Q are normalized to the range [-1, 32767/32768] before quantization. The normalization value is saved to a file whose name is formed by appending ".Normalization.txt" to the File parameter value. The normalization file is intended for use by *ReadBaseBandStudioFile* (algorithm).
5. A marker input is resolved to a zero bit, only if the marker input is zero. The marker bits occupy the least significant bits (LSB) of both integers. The marker name, e.g. Marker15R, indicate which bit and which integer hold the bit value, e.g. Marker15R states the LSB (15) of the first integer (R) is used store Marker15R. The next greater LSB is bit 14.
6. A Browse button is provided at the lower left corner of the properties window to provide a relative path for File. This path is relative to the folder from which the workspace is loaded. If an absolute path is desired, directly enter the full path into the Value column.
7. See *Sink* (algorithm) for a description of the output control parameters.

8. The Base Band Studio format has implementation in ADS and is used in Agilent instrumentation. For more information, see CM BB StudioStreamWrite .

Important Links

1. Learn more about Baseband Studio from Agilent Technologies
2. Also see the following related parts:
   ReadBaseBandStudioFile (algorithm)
   WriteBaseBandStudioFileNormalize (algorithm)

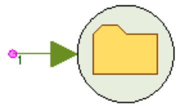
WriteFile Part

Categories: *Sinks* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
WriteN6030File (algorithm)	N6030 Formatted File Writer

WriteN6030File



Description: N6030 Formatted File Writer
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *WriteFile Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range
StartStopOption	Sink collection mode: Auto, Samples, Time	Auto		Enumeration	NO	
SampleStart	Sample number to start data collection	0		Integer	NO	[0:∞)
SampleStop	Sample number to stop data collection	Num_Samples - 1		Integer	NO	[SampleStart:∞)
TimeStart	Time to start data collection	Start_Time	s	Float	NO	[0:∞)
TimeStop	Time to stop data collection	Stop_Time	s	Float	NO	[TimeStart:∞)
FileI	I input file name	file_i.bin		Filename	NO	
FileQ	Q input file name	file_q.bin		Filename	NO	
NormalizeData	Normalize data to avoid clipping: NO, YES	YES		Enumeration	NO	
FullScaleFactor	Full scale factor for peak value	1.0		Float	NO	[0.000030517578125:1]

Input Ports

Port	Name	Description	Signal Type	Optional
1	input	IQ data	complex	NO

Notes/Equations

1. This model writes IQ data into the files *FileI* and *FileQ* using the N6030 format. Data is recorded as 16 bit signed integers.
2. IQ data is represented as a complex value with the I value in the real part and the Q value in the imaginary part.
3. The *StartStopOption* parameter specifies the data collection mode:
   - if set to *Samples* the data collection will start at sample *SampleStart* and stop at *SampleStop*.
   - if set to *Time* the data collection will start at time *TimeStart* and stop at *TimeStop*.
   - if set to *Auto* then all of *SampleStart*, *SampleStop*, *TimeStart*, and *TimeStop* need to be specified and the model will decide at runtime whether *SampleStart*/ *SampleStop* or *TimeStart*/*TimeStop* will be used. The decision is based on whether the input signal has a sampling rate defined. If there is no sampling rate defined (e.g. signal coming out of a *SineGen* (algorithm) source with *SampleRateOption* set to *UnTimed*) then *SampleStart*/*SampleStop* is used. Otherwise, *TimeStart*/*TimeStop* is used.
4. The *FileI* and *FileQ* parameters are used to define the output files for the I and Q data. The files can be specified with a full path name or a relative path name. Relative path names are relative to the directory where the workspace is located.
5. The *NormalizeData* parameter controls whether the data will be clipped or normalized before it is written to the files.
   - if set to *NO*, no normalization is done. Data values outside the range [-*FullScaleFactor*, *FullScaleFactor*] are clipped to the limits of this range. A warning message is also displayed to make the user aware that clipping occured.

- if set to *YES*, no clipping is performed and the data written to the files is a scaled (normalized) version of the input data so that it is always in the range [-*FullScaleFactor*, *FullScaleFactor*]. The scaling factor used is displayed in an information message in the Errors window as well as in the simulation log.

See:
ReadN6030File (algorithm)
WriteBaseBandStudioFile (algorithm)
Sink (algorithm)

Important Links

1. Learn more about N6030 Arbitrary Waveform Generator from Agilent Technologies

Sources

About Sources

Sources are parts with only outputs, i.e. they provide stimulus to a simulation. Every simulation must have at least one source.

In addition to data, sources can optionally associate time with each output sample. To achieve reproducibility, all sources keep time with a 64 bit unsigned integer. The effective time step for the source is represented as a unit quantity with 20 bits of fraction allowing for a precision of 0.95367 10^{-6} (from 2^{-20}) or approximately 1 ppm. The remaining 44 bits allow for 1.7592 10^{13} (from 2^{44} - 1) time steps.

At times, the 64 bit time quantity is converted to a double precision floating point quantity which has only 53 bits of precision. Therefore, the internal time resolution can only be maintained for 8.5899 10^{9} (from 2^{53-20} - 1) time steps and degrades slowly to a minimum precision 1.9531 10^{-3} (from $2^{-(53-44)}$) of one time step.

All source parameter values that are specified in time are rounded to the nearest representation. Parameters that are specified in frequency have their representation in time as a reciprocal are also rounded to the nearest representation. If a source cannot represent a parameter within a 10^{-6} relative error, an error message is generated and the simulation is stopped. Choose a higher effective sample rate (lower effective time step) to resolve this error.

For example, a 1 Hz source has a time step of 1 s.

- Source time range has range [0, 1.7592 10^{13}) seconds or [0, 4.8867 10^{9}) hours or [0, 2.0361 10^{8}) days or [0, 5.57 10^{5}) years.
- A time parameter of 1 - 2^{-21} s is rounded to 1 s and has range [-8.7960 10^{12}, 8.7960 10^{12}] s.
- A frequency parameter of 1/(1 - 2^{-21}) Hz is rounded to 1 Hz and has range [5.6844 10^{-14}, 1.0485 10^{6}] Hz.

Sub classes:

- *Timed Sources* (algorithm)
- *Untimed Sources* (algorithm)

Contents

About Bit Rate Sources

Super class:

- *Untimed Burst Sources* (algorithm)

New bit values are generated at frequency, BitRate, while the part is operating at some sample rate. BitRate cannot exceed the sample rate.

Parameters:

1. BitRate. Default is from the Data Flow Analysis sample rate.

For other parameter descriptions, see *Untimed Burst Sources* (algorithm).

About File Sources

Super class:

- *Untimed Burst Sources* (algorithm)

These parts are classified as file readers where samples are read sequentially from File, decoded and output.
$s_{function}$ = 0 always output the first sample in the file, and so on. For $s_{function}$ definition, see *Untimed Burst Sources* (algorithm).

Parameters:

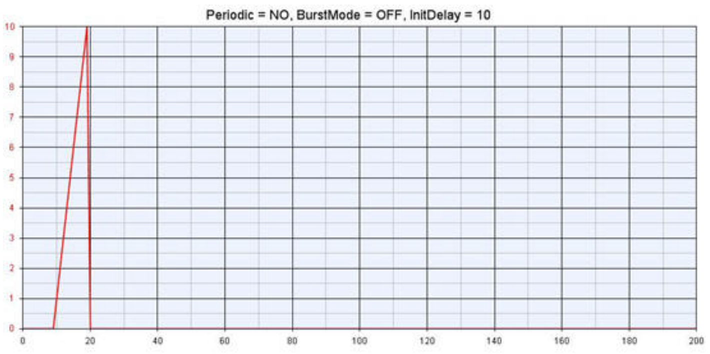
1. File. Default varies by model. A Browse button is provided at the lower left corner of the properties window to provide a relative path for File. This path is relative to the folder from which the workspace is run. If an absolute path is desired, directly enter the full path into the Value column.
2. Periodic. Default is *YES*. If *YES*, the file is reread from the beginning after an end of file condition. Otherwise, zeros are output.

For other parameter descriptions, see *Untimed Burst Sources* (algorithm).

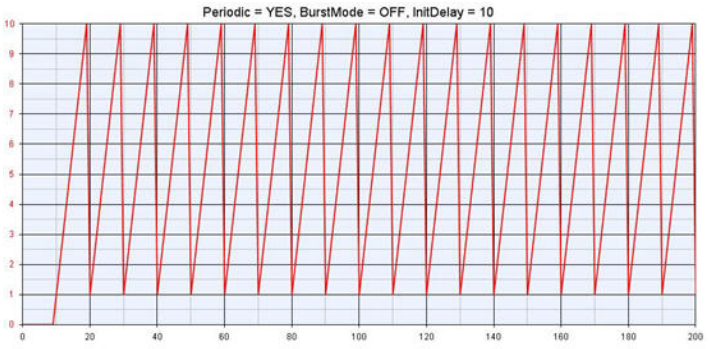
The following table (and plots) summarize the behavior of the file based sources. The plots were generated using the ReadFile model with a file whose contents were the numbers from 1 to 10:

Plot #	Periodic	BurstMode	BurstLength	What is output
1	NO	OFF	Any	file content output once followed by zeros
2	YES	OFF	Any	file content output repeatedly
3	NO	Single	> file content	file content zero padded to BurstLength and output once
4	YES	Single	> file content	file content repeated to BurstLength and output once
5	NO or YES	Single	= file content	file content output once
6	NO or YES	Single	< file content	file content truncated to BurstLength and output once
7	NO	Multiple	> file content	file content zero padded to BurstLength and output in bursts
8	YES	Multiple	> file content	file content repeated to BurstLength and output in bursts
9	NO or YES	Multiple	= file content	file content output in bursts
10	NO or YES	Multiple	< file content	file content truncated to BurstLength and output in bursts

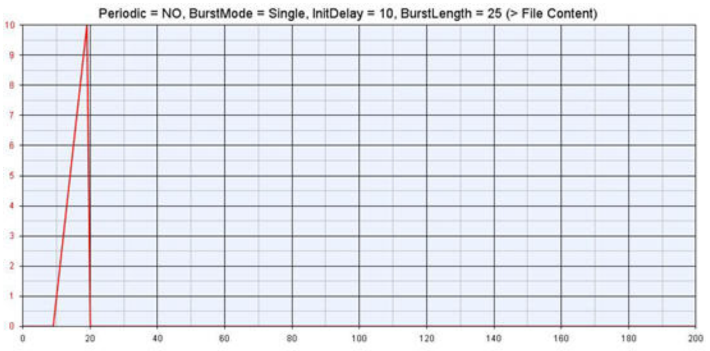
Plot 1



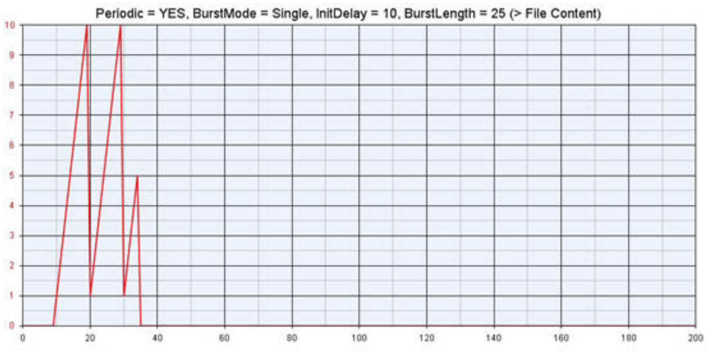
Plot 2



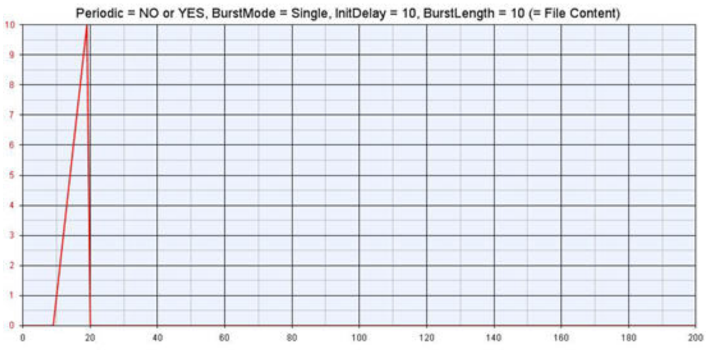
Plot 3



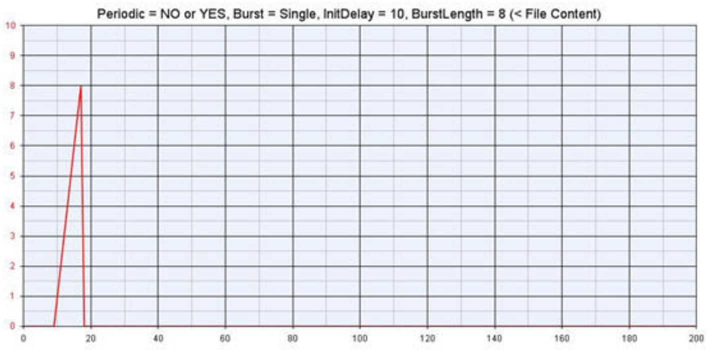
Plot 4



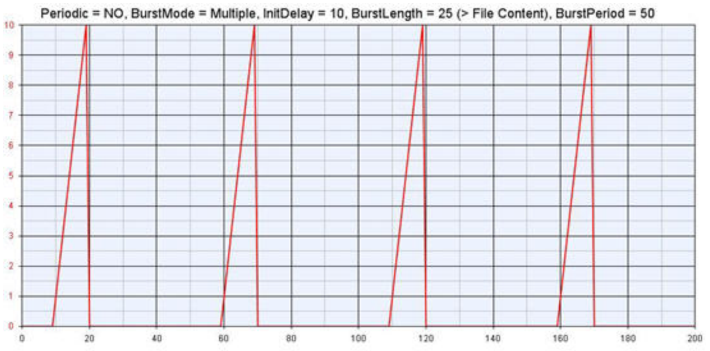
Plot 5



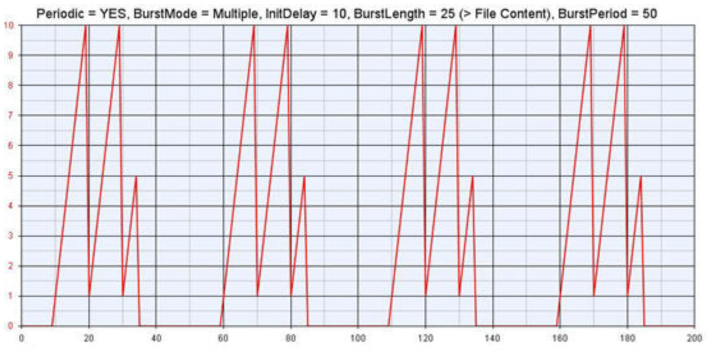
Plot 6



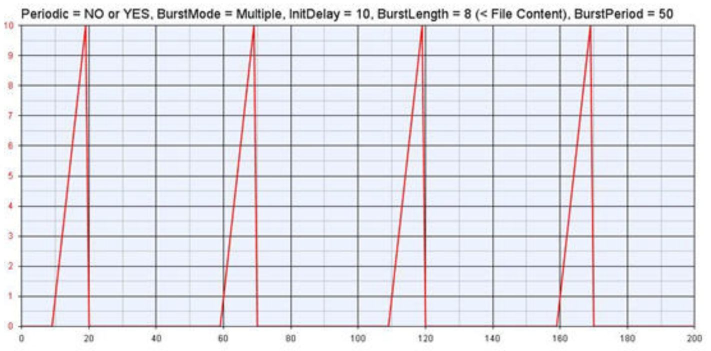
Plot 7



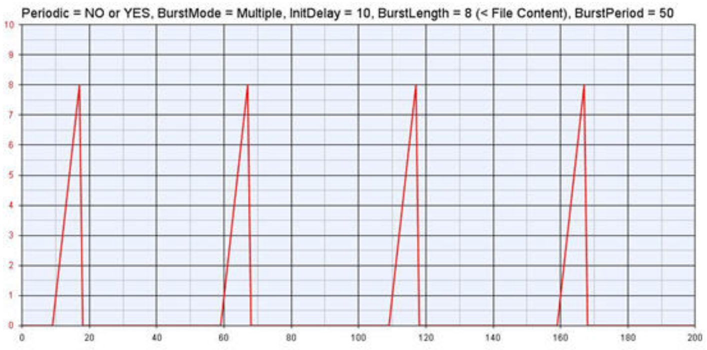
Plot 8



Plot 9



Plot 10



About Time Burst Sources

Super class:

- *Timed Sources* (algorithm).

These parts have burst mode capability. If burst mode is enabled, the output is defined on one (BurstMode is *Single*) or more (BurstMode is *Multiple*) segments of BurstPeriod (BP) seconds. The burst time, t_{burst}, is t_{source} modulo BP. For t_{source} definition, see *Timed Sources* (algorithm).

A signal format exists within a burst period. There can be a delay of BurstDelay (DB) seconds at the beginning and a signal length of BurstLength (BL) seconds after the delay. A quiescent value that is defined for each part is the output for those times outside of the specific signal definition, $DB \le t_{burst} < DB + BL$, but within the burst period.

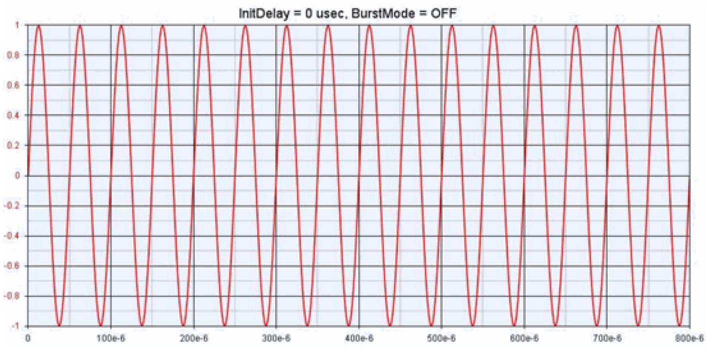
If BurstMode is *OFF*, the specific signal time $t_{function}$ is t_{source}, otherwise $t_{function}$ is $t_{burst} - DB$ when $DB \le t_{burst} < DB + BL$ and unused elsewhere.

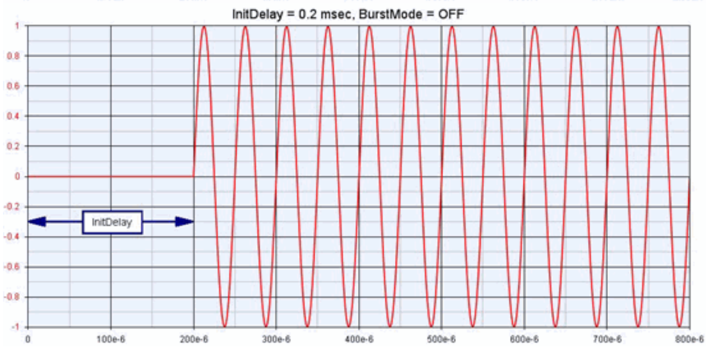
Parameters:

1. BurstMode (B). Default is *OFF*.
2. BurstLength (BL). Default is 100 μs.
3. BurstPeriod (BP). Default is 200 μs.
4. BurstDelay (DB). Default is 0 μs.

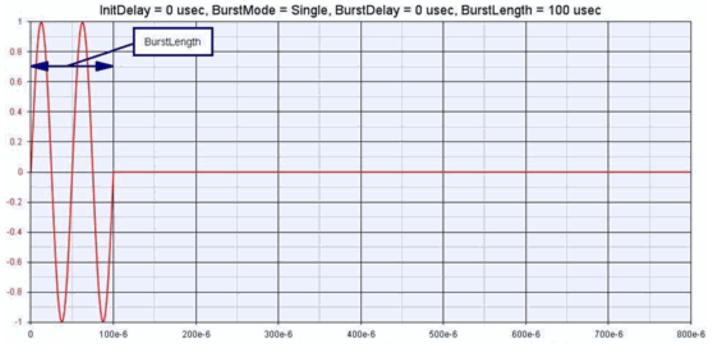
For other parameter descriptions, see *Timed Sources* (algorithm).

The following plots show the output of a *SineGen* (algorithm) source with *Amplitude* = 1 V, *Offset* = 0 V, *Frequency* = 20 kHz, *Phase* = 0°, and different burst configurations.

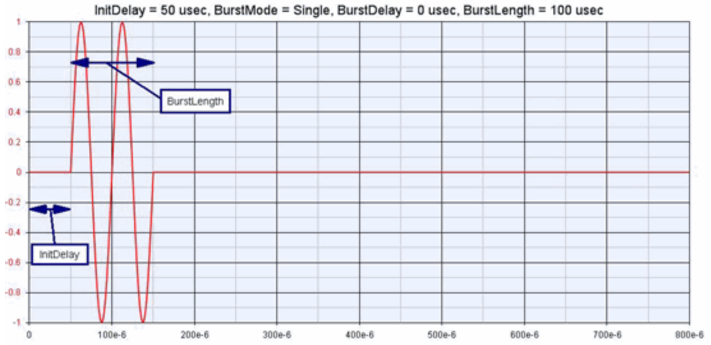




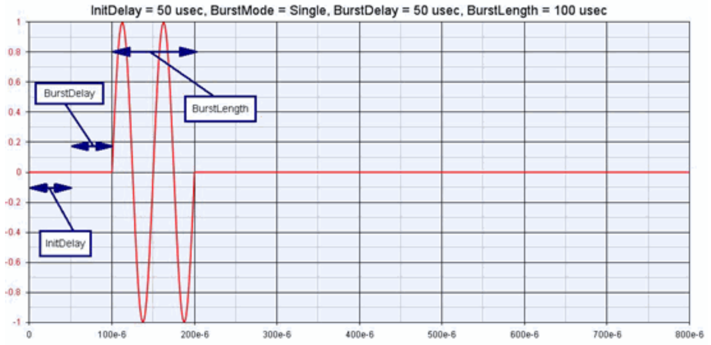
InitDelay = 0 usec, BurstMode = Single, BurstDelay = 0 usec, BurstLength = 100 usec



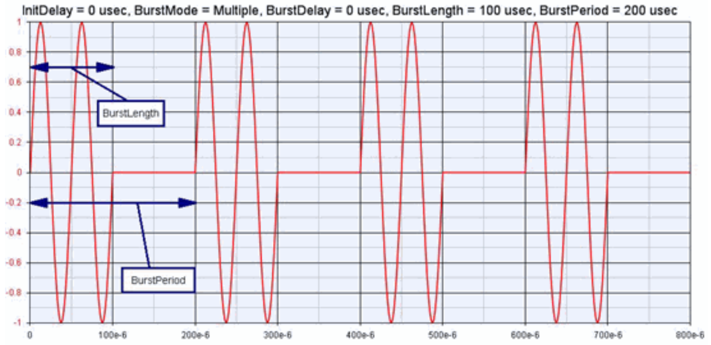
InitDelay = 50 usec, BurstMode = Single, BurstDelay = 0 usec, BurstLength = 100 usec



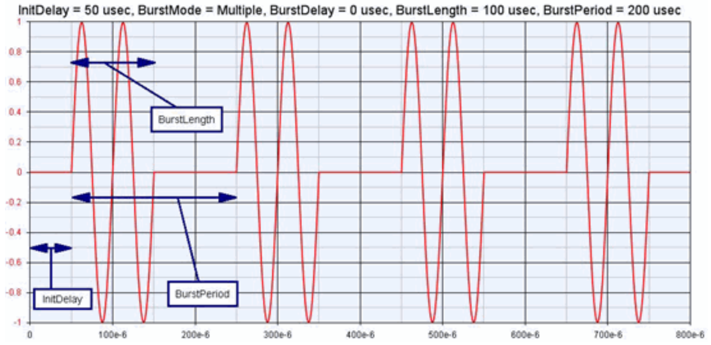
InitDelay = 50 usec, BurstMode = Single, BurstDelay = 50 usec, BurstLength = 100 usec

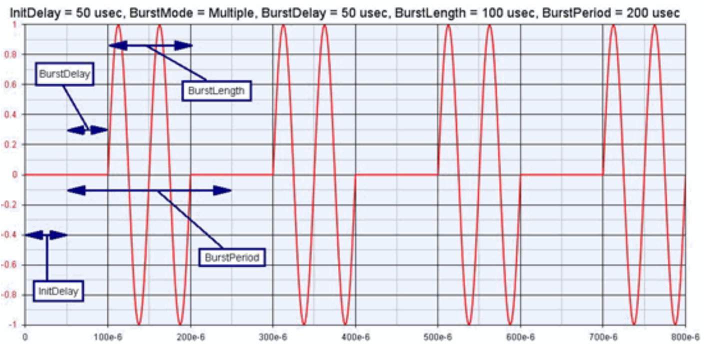


InitDelay = 0 usec, BurstMode = Multiple, BurstDelay = 0 usec, BurstLength = 100 usec, BurstPeriod = 200 usec



InitDelay = 50 usec, BurstMode = Multiple, BurstDelay = 0 usec, BurstLength = 100 usec, BurstPeriod = 200 usec





About Timed Sources

Super class:

- *Sources Category* (algorithm)

Sub classes:

- *Time Burst Sources* (algorithm)
- *Time Swept Sources* (algorithm)

Parts whose signal generation require a sense of time are members of this class. If SampleRateOption is *Timed from Schematic*, then time information comes from the part ports, otherwise the SampleRate parameter is used. If SampleRateOption is set to *Untimed*, the output is simply stripped of time information.

Let t_{sys} be the simulation time which begins at 0 seconds.

Specific signal definition begins at InitialDelay (D) seconds, i.e. $t_{source} \geq 0$ where t_{source} is t_{sys} - D. A fill value defines the signal for $t_{sys} < D$. Typically, a fill value is some form of zero.

The combined signal is sampled from $t_{sys} = 0$ at a fixed sample rate (S), i.e. $t_{sys} = n / S$ for n = 0, 1, 2 ...

Parameters:

1. ShowAdvancedParameters. Default is *NO*.
2. SampleRateOption (SO). Default is *Timed from Schematic*.
3. SampleRate (S). Default is from the Data Flow Analysis sample rate.
4. InitialDelay (D). Default is zero seconds.

About Time Swept Sources

Super class:

- *Timed Sources* (algorithm)

These parts are defined by an instantaneous frequency which is swept from StartFreq (F1) to StopFreq (F2) in a SweepPeriod (SP) time interval. The sweep time, t_{sweep} is derived from t_{source}. t_{sweep} is t_{source} modulo SP. For t_{source} definition, see *Timed Sources* (algorithm).

The instantaneous frequency $F(t_{sweep})$ is:

- $F(t_{sweep}) = F1 + (F2 - F1)\,(t_{sweep} / SP)$, if FSweepType is *linear*.
- $F(t_{sweep}) = F1 + (F2 - F1)\,\log_{10}(1 + 9\,(SP - t_{sweep}) / SP)$, if FSweepType is *log*. $F(t_{sweep})$ is swept linearly in $\log_{10}$ scale.

Parameters:

1. FSweepType (FT). Default is *linear*.
2. StartFreq (F1). Default is 1 KHz.
3. StopFreq (F2). Default is 10 KHz.
4. SweepPeriod (SP). Default is set from the data flow analysis field, Stop Time, thus enabling a single sweep.

For other parameter descriptions, see *Timed Sources* (algorithm).

About Untimed Burst Sources

Super class:

- *Untimed Sources* (algorithm)

Sub classes:

- *Bit Rate Sources* (algorithm)
- *File Sources* (algorithm)

These parts have burst mode capability. If burst mode is enabled, the output is defined on one (BurstMode is *Single*) or more (BurstMode is *Multiple*) BurstPeriod (BP) sample segments. The burst sample number, s_{burst}, is s_{source} modulo BP. For s_{source} definition, see *Untimed Sources* (algorithm).

A signal format exists within a burst period. There can be a delay of BurstDelay (DB) samples at the beginning and a signal length of BurstLength (BL) samples after the delay. A quiescent value that is defined for each part is the output for those sample numbers outside of the specific signal definition, $DB \le s_{burst} < DB + BL$, but within the burst period.

If BurstMode is *OFF*, the specific signal sample number $s_{function}$ is s_{source}, otherwise $s_{function}$ is $s_{burst} - DB$ when $DB \le s_{burst} < DB + BL$ and unused elsewhere.

Parameters:

1. BurstMode. Default is *OFF*.
2. BurstLength (BL). Default is 100 samples.
3. BurstPeriod (BP). Default is 200 samples.
4. BurstDelay (DB). Default is 0 samples.

For other parameter descriptions, see *Untimed Sources* (algorithm).

About Untimed Sources

Super class:

- *Sources Category* (algorithm)

Sub classes:

- *Untimed Burst Sources* (algorithm)

Parts whose signal generation is defined only on sample numbers are members of this class. Time information can be associated with the output, if SampleRateOption is *Timed from Schematic* when time information comes from the part ports, or if SampleRateOption is *Timed from SampleRate* when the SampleRate parameter is used.

Let s_{sys} be the simulation sample number which begins at 0.

Specific signal definition begins at InitialDelay (D) samples, i.e. $s_{source} \ge 0$ where s_{source} is $s_{sys} - D$. A fill value defines the signal for $s_{sys} < D$. Typically, a fill value is some form of zero.

Parameters:

1. ShowAdvancedParameters. Default is *NO*.
2. SampleRateOption (SO). Default is *Timed from Schematic*.
3. SampleRate (S). Default is from the Data Flow Analysis sample rate.
4. InitialDelay (D). Default is 0 sample delays.

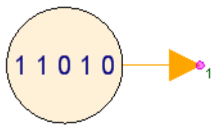
Bits Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
RandomBits (algorithm)	Random Bit Generator
DataPattern (algorithm)	Patterned Data Source
PRBS (algorithm)	Pseudo Random Binary Sequence Generator

DataPattern (Patterned Data Source)



Description: Patterned Data Source
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Bits Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
DataPattern	Data pattern: PN9, PN15, FIX4, _4_1_4_0, _8_1_8_0, _16_1_16_0, _32_1_32_0, _64_1_64_0	PN9		Enumeration	YES		T
BitRate	Output bit rate	Sample_Rate	Hz	Float	NO	[0:SampleRate]	
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		
BurstLength	Burst sample length	100		Integer	YES	[1:∞)	BL
BurstPeriod	Samples from start of one burst to start of next	200		Integer	YES	[1:∞)	BP
BurstDelay	Sample delay within burst before the start of the burst length interval	0		Integer	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Signal Type	Optional
1	output	boolean	NO

Notes/Equations

1. This model generates one of eight specific bit patterns.

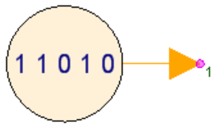
2. For the DataPattern parameter:
   - if *PN9* is selected, a 511-bit pseudo-random test pattern is generated according to CCITT Recommendation O.153
   - if *PN15* is selected, a 32767-bit pseudo-random test pattern is generated according to CCITT Recommendation O.151
   - if *FIX4* is selected, a zero-stream is generated
   - if *x_1_x_0* is selected, where x equals 4, 8, 16, 32, or 64, a periodic bit stream is generated, with the period being 2 × x. In the first half period, x bits are 1s, and in second half period, x bits are 0s.
3. For other parameter descriptions, see *Bit Rate Sources* (algorithm).
4. Fill value is 0.
5. Quiescent value is 0.
6. The bit sequence is restarted when $s_{function}$ is zero. For $s_{function}$ definition, see *Untimed Burst Sources* (algorithm).

See:
PRBS (algorithm)
RandomBits (algorithm)
WalshCode (algorithm)

References

1. CCITT, Recommendation O.151(10/92).
2. CCITT, Recommendation O.153(10/92).

PRBS (Pseudo Random Binary Sequence Generator)



Description: Pseudo Random Binary Sequence Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Bits Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
LFSR_Length	Linear feedback shift register length	12		Integer	YES	[2:31]	L
LFSR_InitState	Linear feedback shift register initial state	1		Integer	YES	[1:2<sup>L</sup>-1]	IS
BitRate	Output bit rate	Sample_Rate	Hz	Float	NO	[0:SampleRate]	
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		
BurstLength	Burst sample length	100		Integer	YES	[1:∞)	BL
BurstPeriod	Samples from start of one burst to start of next	200		Integer	YES	[1:∞)	BP
BurstDelay	Sample delay within burst before the start of the burst length interval	0		Integer	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

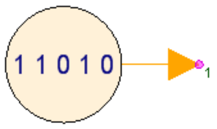
Port	Name	Signal Type	Optional
1	output	boolean	NO

Notes/Equations

1. PRBS generates a pseudo-random bit sequence, using a linear feedback shift register (LFSR).
2. The *LFSR_Length* parameter sets the length of the LFSR as well as selects a maximal-length shift-register code with periodicity $2^{LFSR\_Length} - 1$.
3. The *LFSR_InitState* parameter sets the initial state of the LFSR. Different *LFSR_InitState* values will generate uncorrelated sequences.
4. For other parameter descriptions, see *Bit Rate Sources* (algorithm).
5. Fill value is 0.
6. Quiescent value is 0.
7. The bit sequence is restarted when $s_{function}$ is zero. For $s_{function}$ definition, see *Untimed Burst Sources* (algorithm).

See:
LFSR (algorithm)
DataPattern (algorithm)
RandomBits (algorithm)
WalshCode (algorithm)

RandomBits (Random Bit Generator)



Description: Random Bit Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Bits Part* (algorithm)

Model Parameters

Name	**Description**	**Default**	**Units**	**Type**	**Runtime Tunable**	**Range**	**Symbol**
ProbOfZero	Probability of zero bit value	0.5		Float	YES	[0:1]	P
BitRate	Output bit rate	Sample_Rate	Hz	Float	NO	[0:SampleRate]	
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		
BurstLength	Burst sample length	100		Integer	YES	[1:∞)	BL
BurstPeriod	Samples from start of one burst to start of next	200		Integer	YES	[1:∞)	BP
BurstDelay	Sample delay within burst before the start of the burst length interval	0		Integer	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	**Name**	**Signal Type**	**Optional**
1	output	boolean	NO

Notes/Equations

1. This model generates a random bit sequence, in which the probability of a 0 bit is *ProbOfZero* and the probability of a 1 bit is *1 - ProbOfZero*.
2. For other parameter descriptions, see *Bit Rate Sources* (algorithm).
3. Fill value is 0.
4. Quiescent value is 0.
5. The repeatability of the bit sequence generated by this model can be controlled by the *Repeatable Random Sequences* check box in the *Option tab of the DF Analysis dialog* (sim).

See:
LFSR (algorithm)
DataPattern (algorithm)
PRBS (algorithm)
WalshCode (algorithm)

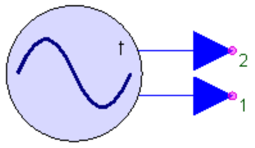
ChirpGen Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
ChirpGen (algorithm)	Frequency Chirp Generator

ChirpGen



Description: Frequency Chirp Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *ChirpGen Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
Amplitude	Peak sine wave voltage amplitude	1	V	Float	YES	(-∞:∞)	A
Offset	DC offset voltage	0	V	Float	YES	(-∞:∞)	DC
StartFreq	Start frequency	1e3	Hz	Float	YES	(0:SampleRate/4]	F1
StopFreq	Stop frequency	10e3	Hz	Float	YES	(0:SampleRate/4]	F2
Phase	Phase	0	deg	Float	YES	(-∞:∞)	PH
SweepPeriod	Time period to complete a sweep	Stop_Time	s	Float	YES	[1/SampleRate:∞)	SP
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO
2	frequency	real	NO

Notes/Equations

1. The ChirpGen model outputs a sine wave with *Amplitude* , *Phase* and *Offset* and swept in the frequency domain from *StartFreq* to *StopFreq* over the time interval *SweepPeriod* .
2. Let $F(t_{sweep})$ be the instantaneous time domain sweep frequency defined in *Time Swept Sources* (algorithm).
3. The waveform value, $A \sin(2 \pi (F(t_{sweep}) t_{sweep} + PH / 360)) + DC$, is output.
4. $F(t_{sweep})$ is output at the frequency port in Hertz. The maximum frequency domain spectral frequency is obtained when the time domain $F(t_{sweep})$ value is at its maximum value of ((*StopFreq* + *StartFreq*) / 2).
5. For other parameter descriptions, see *Time Swept Sources* (algorithm).
6. Fill value is 0.

See:
ComplexExpGen (algorithm)
SineGen (algorithm)
SineSweepGen (algorithm)

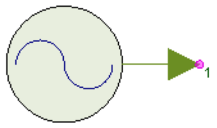
ComplexExpGen Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
ComplexExpGen (algorithm)	Complex Exponential Source

ComplexExpGen (Complex Exponential Source)



Description: Complex Exponential Source
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *ComplexExpGen Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
Amplitude	Peak sine wave voltage amplitude	1	V	Float	YES	(-∞:∞)	A
Offset	DC offset complex voltage	0	V	Complex number	YES		DC
Frequency	Frequency	5e3	Hz	Float	YES	[0:SampleRate/4]	F
Phase	Phase	0	deg	Float	YES	(-∞:∞)	PH
QuadraturePolarity	Quadrature polarity: normal, inverted	normal		Enumeration	YES		QP
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		B
BurstLength	Burst time length	100e-6	s	Float	YES	[1/SampleRate:∞)	BL
BurstPeriod	Time period from start of one burst to start of next	200e-6	s	Float	YES	[1/SampleRate:∞)	BP
BurstDelay	Time delay within burst before the start of the BurstLength interval	0	s	Float	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Signal Type	Optional
1	output	complex	NO

Notes/Equations

1. The function, $A\ e^{Q\ (2\ \pi\ (F\ t_{function} + PH\ /\ 360))}$ + DC, is output. Q is j for normal Quadrature Polarity, otherwise Q is -j. For $t_{function}$ definition, see *Time Burst Sources* (algorithm).
2. For other parameter descriptions, see *Time Burst Sources* (algorithm).
3. Fill value is 0.
4. Quiescent value is DC.

See:
Oscillator Nonlinear Gain
SineGen (algorithm)
SineSweepGen (algorithm)

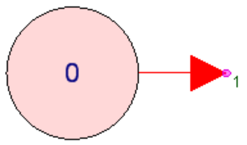
Const Part

Categories: *C++ Code Generation* (algorithm), *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
Const (algorithm)	Constant Generator
ConstFxp (hardware)	Fixed Point Constant

Const (Constant Generator)



Description: Constant Generator
Domain: Timed
C++ Code Generation Support: YES (see Note)
Associated Parts: *Const Part* (algorithm)

Model Parameters

Name	**Description**	**Default**	**Units**	**Type**	**Runtime Tunable**	**Range**	**Symbol**
Value	Value	0		None	YES		V
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D

Output Ports

Port	**Name**	**Signal Type**	**Optional**
1	output	anytype	NO

Note
This model does not support C++ code generation if the port type is resolved to *variant* (sim) or if *SampleRateOption* is NOT set to *UnTimed*.

Notes/Equations

1. Output a constant signal with value given by the Value parameter. For discussion on the variant type, see *Variant* (sim).
2. For other parameter descriptions, see *Untimed Sources* (algorithm).
3. Fill value is 0.

See:
Identity_M (algorithm)
IdentityCx_M (algorithm)
Diagonal_M (algorithm)
DiagonalCx_M (algorithm)

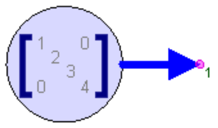
Diagonal_M Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Diagonal_M (algorithm)	Diagonal Matrix Generator
DiagonalCx_M (algorithm)	Output matrix diagonal elements

Diagonal_M (Diagonal Matrix Generator)



Description: Diagonal Matrix Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Diagonal M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
DiagonalElements	Output matrix diagonal elements	[1, 2]		Floating point array	NO		E
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D

Output Ports

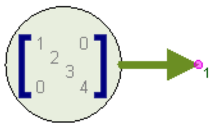
Port	Name	Signal Type	Optional
1	output	real matrix	NO

Notes/Equations

1. Diagonal_M is a constant square matrix source whose dimensions are specified by the RowsCols parameter.
2. The constant matrix is formed by inserting values from the real DiagonalElements array parameter into the diagonal elements beginning at element [1,1]. Zeroes are inserted at the off diagonal elements.
3. The number of elements in DiagonalElements must be at least RowCols.
4. For other parameter descriptions, see *Untimed Sources* (algorithm).
5. Fill value is a real matrix zero.

See:
DiagonalCx_M (algorithm)
Identity_M (algorithm)
IdentityCx_M (algorithm)
Const (algorithm)

DiagonalCx_M (Output matrix diagonal elements)



Description: Output matrix diagonal elements
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Diagonal M Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
DiagonalElements	Output matrix diagonal elements	[1, j]		Complex array	NO		E
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D

Output Ports

Port	Name	Signal Type	Optional
1	output	complex matrix	NO

Notes/Equations

1. DiagonalCx_M is a constant square matrix source whose dimensions are specified by the RowsCols parameter.
2. The constant matrix is formed by inserting values from the complex DiagonalElements array parameter into the diagonal elements beginning at element [1,1]. Zeroes are inserted at the off diagonal elements.
3. The number of elements in DiagonalElements must be at least RowCols.
4. For other parameter descriptions, see *Untimed Sources* (algorithm).
5. Fill value is a complex matrix zero.

See:
Diagonal_M (algorithm)
Identity_M (algorithm)
IdentityCx_M (algorithm)
Const (algorithm)

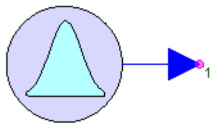
GaussianNoiseGen Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
GaussianNoiseGen (algorithm)	Gaussian Noise Generator

GaussianNoiseGen (Gaussian Noise Generator)



Description: Gaussian Noise Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *GaussianNoiseGen Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
NDensity	Noise power spectral density	4.00388587e-21	W	Float	YES	[0:∞)	ND
RefR	Reference resistance	50	ohm	Float	NO	(0:∞)	
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		B
BurstLength	Burst time length	100e-6	s	Float	YES	[1/SampleRate:∞)	BL
BurstPeriod	Time period from start of one burst to start of next	200e-6	s	Float	YES	[1/SampleRate:∞)	BP
BurstDelay	Time delay within burst before the start of the BurstLength interval	0	s	Float	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO

Notes/Equations

1. Output additive white Gaussian noise (AWGN) that is characterized by the NDensity parameter over the bandwidth specified by the SampleRate parameter into RefR ohms.
2. The Gaussian noise has zero mean and standard deviation of √ {NDensity × (SampleRate / 2) × RefR}.
3. NDensity is the noise density in Watts/Hz and is related to temperature T in Kelvin as k × T where k is the Boltzmann constant (1.3806504e-23). At the standard system temperature of 290 Kelvin (16.85 Celsius), the NDensity is 4.00388587e-21 Watts/Hz (-173.975 dBm/Hz).
4. For other parameter descriptions, see *Time Burst Sources* (algorithm).

5. Fill value is 0.
6. Quiescent value is 0.

See:
IID_Gaussian (algorithm)
IID_Uniform (algorithm)

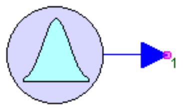
IID_Gaussian Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
IID_Gaussian (algorithm)	IID Gaussian Noise Waveform

IID_Gaussian (IID Gaussian Noise Waveform)



Description: IID Gaussian Noise Waveform
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *IID Gaussian Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
StdDev	Standard deviation	1	V	Float	YES	(-∞:∞)	SD
Offset	Offset value	0	V	Float	YES	(-∞:∞)	O
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		
BurstLength	Burst sample length	100		Integer	YES	[1:∞)	BL
BurstPeriod	Samples from start of one burst to start of next	200		Integer	YES	[1:∞)	BP
BurstDelay	Sample delay within burst before the start of the burst length interval	0		Integer	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO

Notes/Equations

1. Identically independently distributed variates with Gaussian distribution as specified by mean from the Offset parameter and standard deviation from the StdDev parameter are output. The output is commonly referenced as additive white Gaussian noise (AWGN).
2. For other parameter descriptions, see *Untimed Burst Sources* (algorithm).
3. Fill value is 0.
4. Quiescent value is 0.

See:
IID_Uniform (algorithm)
GaussianNoiseGen (algorithm)

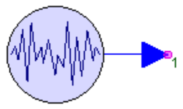
IID_Uniform Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
IID_Uniform (algorithm)	IID Uniform Noise Waveform

IID_Uniform (IID Uniform Noise Waveform)



Description: IID Uniform Noise Waveform
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *IID Uniform Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
LoLevel	Low voltage level	0	V	Float	YES	(-∞:∞)	VL
HiLevel	High voltage level	1	V	Float	YES	(LoLevel:∞)	VH
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		
BurstLength	Burst sample length	100		Integer	YES	[1:∞)	BL
BurstPeriod	Samples from start of one burst to start of next	200		Integer	YES	[1:∞)	BP
BurstDelay	Sample delay within burst before the start of the burst length interval	0		Integer	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO

Notes/Equations

1. Identically independently distributed variates with uniform distribution between the LoLevel and HiLevel parameters is output.
2. For other parameter descriptions, see *Untimed Burst Sources* (algorithm).
3. Fill value is 0.
4. Quiescent value is 0.

See:
IID_Gaussian (algorithm)
GaussianNoiseGen (algorithm)

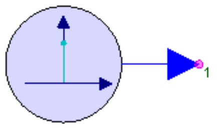
Impulse Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Impulse (algorithm)	Impulse Waveform

Impulse (Impulse Waveform)



Description: Impulse Waveform
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Impulse Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
Level	Impulse level	1	V	Float	YES	(-∞:∞)	L
ScaleBySampleRate	Scale impulse level by sample rate: NO, YES	NO		Enumeration	NO		
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		B
BurstLength	Burst time length	100e-6	s	Float	YES	[1/SampleRate:∞)	BL
BurstPeriod	Time period from start of one burst to start of next	200e-6	s	Float	YES	[1/SampleRate:∞)	BP
BurstDelay	Time delay within burst before the start of the BurstLength interval	0	s	Float	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO

Notes/Equations

1. *Level* or *Level* × SR is output for t_{function} in [0, 1/S) where SR is the effective output sample rate. Zero is output everywhere else.
   - If *SampleRateOption* is *Timed from Sample Rate* or *Timed from Schematic* and *ScaleBySampleRate* is *YES*, then *Level* is multiplied by SR. This normalizes the impulse to have unity area, i.e. a delta function *δ(InitialDelay)*. This option should be used when looking at the frequency response of filters.
   - If *SampleRateOption* is *UnTimed*, *ScaleBySampleRate* is ignored.
2. Jitter may occur, if the time parameters are not expressed as an integer multiple of 1/S.
   For other parameter descriptions, see *Time Burst Sources* (algorithm).

3.
4. Fill value is 0.
5. Quiescent value is 0.

See:
PulseGen (algorithm)
SquareGen (algorithm)
SquareSweepGen (algorithm)

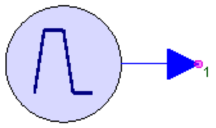
PulseGen Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
PulseGen (algorithm)	Pulse Waveform Generator

PulseGen (Pulse Waveform Generator)



Description: Pulse Waveform Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *PulseGen Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
LoLevel	Low voltage level	0	V	Float	YES	(-∞:∞)	VL
HiLevel	High voltage level	1	V	Float	YES	(LoLevel:∞)	VH
Period	Pulse waveform period	200e-6	s	Float	YES	[2/SampleRate:∞)	PP
Phase	Phase	0	deg	Float	YES	(-∞:∞)	PH
PulseWidth	Pulse width at 50% waveform levels	100e-6	s	Float	YES	[EdgeTime:Period-EdgeTime]	PW
EdgeTime	Rising and falling edge times (0% to 100% values)	50e-6	s	Float	YES	[1/SampleRate:Period/2]	E
Polarity	Signal polarity: normal, inverted	normal		Enumeration	YES		P
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		B
BurstLength	Burst time length	100e-6	s	Float	YES	[1/SampleRate:∞)	BL
BurstPeriod	Time period from start of one burst to start of next	200e-6	s	Float	YES	[1/SampleRate:∞)	BP
BurstDelay	Time delay within burst before the start of the BurstLength interval	0	s	Float	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO

Notes/Equations

1. A pulse W(t) with a rise and a fall interval is defined on t from [0, PP).
2. The output W(t) for normal Polarity is:
    - VL + (VH - VL) (t / E), for $0 \leq t < E$
    - VH, for $E \leq t < PW$
    - VH - (VL - VH) (t - PW) / E, for $PW \leq t < PW + E$
    - VL, for $PW + E \leq t < PP$
3. For inverted Polarity, the pulse output begins at VH and grows downward to VL, and so on.
4. For simulation time $t_{function}$, t is ($t_{function}$ + (PH' / 360) PP) modulo PP, where PH' is ((PH modulo 360) + 360) modulo 360.
   If zero PH, t is $t_{function}$ modulo PP. For $t_{function}$ definition, see *Time Burst Sources* (algorithm).
5. For other parameter descriptions, see *Time Burst Sources* (algorithm).
6. Fill value is 0.
7. The quiescent value is:
    - VL, if Polarity (P) is normal
    - VH, otherwise

See:
Impulse (algorithm)
SquareGen (algorithm)
SquareSweepGen (algorithm)

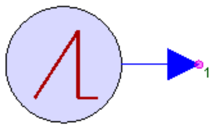
RampGen Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
RampGen (algorithm)	Ramp Waveform Generator

RampGen (Ramp Waveform Generator)



Description: Ramp Waveform Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *RampGen Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
LoLevel	Low voltage level	0	V	Float	YES	(-∞:∞)	VL
HiLevel	High voltage level	1	V	Float	YES	(LoLevel:∞)	VH
Frequency	Waveform frequency	5e3	Hz	Float	YES	[0:SampleRate/4]	F
Phase	Phase	0	deg	Float	YES	(-∞:∞)	PH
Symmetry	Symmetry in percent	100		Float	YES	[0:100]	RS
Polarity	Signal polarity: normal, inverted	normal		Enumeration	YES		P
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		B
BurstLength	Burst time length	100e-6	s	Float	YES	[1/SampleRate:∞)	BL
BurstPeriod	Time period from start of one burst to start of next	200e-6	s	Float	YES	[1/SampleRate:∞)	BP
BurstDelay	Time delay within burst before the start of the BurstLength interval	0	s	Float	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO

Notes/Equations

1. A triangular waveform W(t) is defined on t from [0, 1/F).
2. Let t $_{inflection}$ be the time of the waveform inflection. t $_{inflection}$ is RS / 100 / F.

3. For normal Polarity, the output W(t) is:
   $VL + (VH - VL) (t / t_{inflection})$ for $0 \leq t < t_{inflection}$

   $VH + (VL - VH) ((t - t_{inflection}) / ((1 / F) - t_{inflection}))$ for $t_{inflection} \leq t < 1 / F$.
4. For inverted polarity, the waveform begins with VH and goes to VL.
5. For simulation time, $t_{function}$, t is $(t_{function} + PH' / 360 / F)$ modulo $(1 / F)$, where

   PH' = ((PH modulo 360) + 360) modulo 360.
   If zero PH, t is $t_{function}$ modulo (1 / F). For $t_{function}$ definition, see *Time Burst Sources* (algorithm).
6. For other parameter descriptions, see *Time Burst Sources* (algorithm).
7. Fill value is 0.
8. Quiescent value is (VL + VH) / 2.

See:
Ramp (algorithm)
RampSweepGen (algorithm)

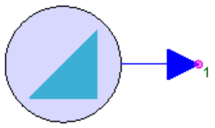
Ramp Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Ramp (algorithm)	Ramp Waveform

Ramp (Ramp Waveform)



Description: Ramp Waveform
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Ramp Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
StepPerSample	Step value per simulation sample	1		Float	YES	(-∞:∞)	SS
InitialValue	Initial ramp voltage value	0		Float	YES	(-∞:∞)	I
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO

Notes/Equations

1. A ramp waveform is generated.
2. The waveform begins with value I at $s_{function} = 0$ and increments by SS for each sample thereafter.
3. For other parameter descriptions, see *Untimed Sources* (algorithm).
4. Fill value is 0.
5. The default parameters counts samples from 0.

See:
RampGen (algorithm)
RampSweepGen (algorithm)

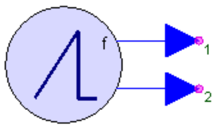
RampSweepGen Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
RampSweepGen (algorithm)	Ramp Sweep Waveform Generator

RampSweepGen (Ramp Sweep Waveform Generator)



Description: Ramp Sweep Waveform Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *RampSweepGen Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
LoLevel	Low voltage level	0	V	Float	YES	(-∞:∞)	VL
HiLevel	High voltage level	1	V	Float	YES	(LoLevel:∞)	VH
FSweepType	Frequency sweep type: linear, log	linear		Enumeration	YES		FT
StartFreq	Start frequency	1e3	Hz	Float	YES	(0:SampleRate/4]	F1
StopFreq	Stop frequency	10e3	Hz	Float	YES	(0:SampleRate/4]	F2
Phase	Phase	0	deg	Float	YES	(-∞:∞)	PH
SweepPeriod	Time period to complete a sweep	Stop_Time	s	Float	YES	[1/SampleRate:∞)	SP
Symmetry	Symmetry in percent	100		Float	YES	[0:100]	RS
Polarity	Signal polarity: normal, inverted	normal		Enumeration	YES		P
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO
2	frequency	real	NO

Notes/Equations

1. Let F(t $_{sweep}$) be the instantaneous sweep frequency defined in *Time Swept Sources* (algorithm).
2. A triangular waveform W(t') is defined on t' from [0, 1 / F(t $_{sweep}$)). Let t $_{inflection}$ be RS / 100 / F(t $_{sweep}$).
3. Let t be (t $_{sweep}$ + (PH' / 360 / F(t $_{sweep}$))) modulo (1 / F(t $_{sweep}$)), where PH' = ((PH modulo 360) + 360) modulo 360.

If PH is zero, t is t_{sweep} modulo (1 / F(t_{sweep})).

4. For *normal* Polarity, the output W(t) is:
   VL + (VH - VL) t / $t_{inflection}$, if $0 \le t < t_{inflection}$

   VH + (VL - VH) ((t - $t_{inflection}$) / ((1 / F(t_{sweep})) - $t_{inflection}$)), if $t_{inflection} \le t < 1$ / F(t_{sweep})
5. For *inverted* Polarity, the output W(t) is:
   VH + (VL - VH) t / $t_{inflection}$, if $0 \le t < t_{inflection}$

   VL + (VH - VL) ((t - $t_{inflection}$) / ((1 / F(t_{sweep})) - $t_{inflection}$)), if $t_{inflection} \le t < 1$ / F(t_{sweep})
6. F(t_{sweep}) is output at the frequency port in Hertz.
7. For other parameter descriptions, see *Time Swept Sources* (algorithm).
8. Fill value is 0.

See:
Ramp (algorithm)
RampGen (algorithm)

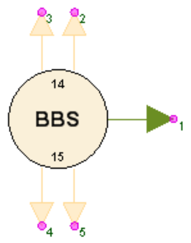
ReadBaseBandStudioFile Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
ReadBaseBandStudioFile (algorithm)	Base Band Studio Format File Reader

ReadBaseBandStudioFile



Description: Base Band Studio Format File Reader
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *ReadBaseBandStudioFile Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
File	Input file name	file.bin		Filename	NO		F
Periodic	Repeat file data when end of file is reached: NO, YES	YES		Enumeration	YES		P
MarkerFormat	marker bits: Zero Markers (16 Bit IQ), Two Markers (15 Bit IQ), Four Markers (14 Bit IQ)	Zero Markers (16 Bit IQ)		Enumeration	NO		M
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		
BurstLength	Burst sample length	100		Integer	YES	[1:∞)	BL
BurstPeriod	Samples from start of one burst to start of next	200		Integer	YES	[1:∞)	BP
BurstDelay	Sample delay within burst before the start of the burst length interval	0		Integer	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Description	Signal Type	Optional
1	IQ	IQ data	complex	NO
2	Marker14R	Real Bit 14 Marker	int	YES
3	Marker14I	Imag Bit 14 Marker	int	YES
4	Marker15R	Real Bit 15 Marker	int	YES
5	Marker15I	Imag Bit 15 Marker	int	YES

Notes/Equations

1. This model reads complex IQ data and, optionally, markers in the Base Band Studio format from a file named by the File parameter.
2. Each Base Band Studio sample consists of a pair of 16 bit signed integers. The first integer hold the real parts while the second integer hold the imaginary parts.
3. The MarkerFormat parameter specify how IQ data are encoded and how many markers are output. IQ data may have 14, 15 or 16 bit quantizations. With 14 bit quantization, there are four marker values, i.e. Marker14I, Marker14R, Marker15R and Marker15I. With 15 bit quantization, there are two marker values, Marker15R and Marker15I. With 16 bit quantization, there are no marker values.
4. IQ is represented as a complex value with the I value in the imaginary part and the Q value in the real part. Both I and Q are normalized such that the absolute value of either never exceed unity. However, the IQ can be rescaled with a value that is read from a normalization file. The file name is formed from the Base Band Studio file name appended with ".Normalization.txt". This file can be editted. If such file is not available in the same folder as File, there is no scaling.
5. Marker15R is taken from the least significant bit (LSB) of the first 16 bit integer. Marker15I is taken from the LSB of the second 16 bit integer. Similarly, Marker14R and Marker14I are formed from the next higher LSB. These markers output zero or one. Zeros are output for a no marker value.
6. The Base Band Studio format has implementation in ADS and is used in Agilent instrumentation. For more information, see CM BB StudioStreamRead .
7. For other parameter descriptions, see *File Sources* (algorithm).
8. Fill values are complex zeroes for IQ and integer zeros for the markers.
9. Quiescent values are complex zeroes for IQ and integer zeros for the markers.

Important Links

1. Learn more about Baseband Studio from Agilent Technologies
2. Also see the following related models:
   ReadFile (algorithm)
   ReadN5106AFile (algorithm)
   ReadN6030File (algorithm)
   ReadSignalStudioFile (algorithm)
   WriteBaseBandStudioFile (algorithm)
   WriteBaseBandStudioFileNormalize (algorithm)

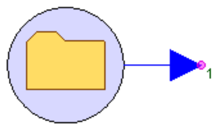
ReadFile Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
ReadFile (algorithm)	Waveform Output from File
ReadN5106AFile (algorithm)	N5106A Format File Reader
ReadN6030File (algorithm)	N6030 Format File Reader
ReadSignalStudioFile (algorithm)	Time domain signal generator with signalstudio encrypted file based data

ReadFile (Waveform Output from File)



Description: Waveform Output from File
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *ReadFile Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
File	Input file name	file.txt		Filename	NO		F
Periodic	Repeat file data when end of file is reached: NO, YES	YES		Enumeration	YES		P
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		
BurstLength	Burst sample length	100		Integer	YES	[1:∞)	BL
BurstPeriod	Samples from start of one burst to start of next	200		Integer	YES	[1:∞)	BP
BurstDelay	Sample delay within burst before the start of the burst length interval	0		Integer	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO

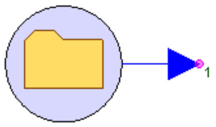
Notes/Equations

1. This model reads float data in ASCII text form from a file named by the File parameter.
2. Float data are separated by spaces or a new line.
3. For other parameter descriptions, see *File Sources* (algorithm).

4. Fill value is 0.
5. Quiescent value is 0.

See:
ReadN5106AFile (algorithm)
ReadN6030File (algorithm)
ReadSignalStudioFile (algorithm)
ReadBaseBandStudioFile (algorithm)
Sink (algorithm)

ReadN5106AFile



Description: N5106A Format File Reader
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *ReadFile Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
File	Input file name	myn5106a.bin		Filename	NO		F
Periodic	Repeat file data when end of file is reached: NO, YES	YES		Enumeration	YES		P
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		
BurstLength	Burst sample length	100		Integer	YES	[1:∞)	BL
BurstPeriod	Samples from start of one burst to start of next	200		Integer	YES	[1:∞)	BP
BurstDelay	Sample delay within burst before the start of the burst length interval	0		Integer	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Signal Type	Optional
1	output	complex	NO

Notes/Equations

1. This model reads float data in N5106A binary format from a file named by the File parameter.
2. N5106A is a binary waveform format that is targeted for use by the Agilent N5106A PXB MIMO Receiver Tester.
3. For other parameter descriptions, see *File Sources* (algorithm).
4. Fill value is 0.
5. Quiescent value is 0.

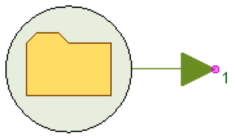
See:
ReadFile (algorithm)
ReadN6030File (algorithm)

ReadSignalStudioFile (algorithm)
ReadBaseBandStudioFile (algorithm)
Sink (algorithm)

Important Links

1. The main purpose for this file reader is to read back the I/Q data written into a N5106A file by the *Sink* (algorithm)
2. Learn more about the PXB/N5106A instrument from Agilent Technologies.
3. To download the N5106A PXB MIMO Receiver Tester SW to run on your PC in simulated mode, visit the download site from Agilent Technologies.
4. Also see the related part: *SignalDownloader_N5106A* (algorithm)

ReadN6030File



Description: N6030 Format File Reader
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *ReadFile Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
FileI	Input file name	file_i.bin		Filename	NO		FI
FileQ	Input file name	file_q.bin		Filename	NO		FQ
Periodic	Repeat file data when end of file is reached: NO, YES	YES		Enumeration	YES		P
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		
BurstLength	Burst sample length	100		Integer	YES	[1:∞)	BL
BurstPeriod	Samples from start of one burst to start of next	200		Integer	YES	[1:∞)	BP
BurstDelay	Sample delay within burst before the start of the burst length interval	0		Integer	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Description	Signal Type	Optional
1	output	IQ data	complex	NO

Notes/Equations

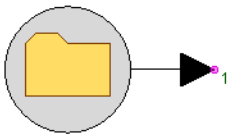
1. This model reads I and Q data from the N6030 formatted files *FileI* and *FileQ* to form a complex IQ output.
2. If one file is longer than the other, the shorter file will be zero padded. A warning is generated for this condition.
3. IQ data is represented as a complex value with the I value in the real part and the Q value in the imaginary part.
4. For other parameter descriptions, see *File Sources* (algorithm).
5. Fill value is 0.0 + j·0.0.
6. Quiescent value is 0.0 + j·0.0.

See:
ReadFile (algorithm)
ReadN5106AFile (algorithm)
ReadSignalStudioFile (algorithm)
ReadBaseBandStudioFile (algorithm)
WriteN6030File (algorithm)

Important Links

1. Learn more about N6030 Arbitrary Waveform Generator from Agilent Technologies

ReadSignalStudioFile



Description: Time domain signal generator with signalstudio encrypted file based data
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *ReadFile Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
WaveformFile	Choose to either use one of the waveform files from SystemVue or provide a waveform file.: Built In, User Defined	Built In		Enumeration	NO		
File	input file name	esg.wfm		Filename	NO		
BuiltInFile	Interal Signal Studio waveforms: 2TONE, 12TONE, CDMA2K_9CHAN, CDMA2K_PILOT, EDGE_1C_BURST, EDGE_1C_CONT, GSM_1C_BURST, GSM_1C_CONT, QAM16, QPSK, WCDMA_1DPCH, WCDMA_TM1_64DPCH_1C, WCDMA_TM1_64DPCH_4C, WCDMA_TM4, WIMAX_10MHZ_64QAM, WIMAX_10MHZ_QPSK, WLAN20MHZ54M64QAM	WLAN20MHZ54M64QAM		Enumeration	NO		
Periodic	Repeat file data when end of file is reached: NO, YES	YES		Enumeration	NO		P
SetCarrierFrequency	Set RF carrier frequency: NO, YES	NO		Enumeration	NO		
RFCarrier	RF carrier frequency (used to overide what is specified in the waveform file)	1.0e9	Hz	Float	NO	(0:∞)	f_c
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: Timed from source data, Timed from SampleRate, Timed from Schematic	Timed from source data		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InterpolationType	Interpolation method: Sample and hold, Linear, Lagrange	Sample and hold		Enumeration	NO		
InterpolationOrder	Lagrange interpolation order	4		Integer	NO	[2:∞)	
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		B
BurstLength	Burst time length	100e-6	s	Float	YES	[1/SampleRate:∞)	BL
BurstPeriod	Time period from start of one burst to start of next	200e-6	s	Float	YES	[1/SampleRate:∞)	BP
BurstDelay	Time delay within burst before the start of the BurstLength interval	0	s	Float	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Signal Type	Optional
1	output	envelope	NO

⚠ Currently **without** **SystemVue's LTE or WiMax license**, **no** waveform files created by Signal Studio software can be read; **with SystemVue's LTE or WiMax license**, the corresponding waveform file (i.e. LTE or WiMax) created by Signal Studio software can be read.

⚠ With **WaveformFile = Built In**, there are 17 waveforms available for playback.

⚠ The ReadSignalStudioFile model can read all Agilent Signal Studio waveform format files created by other Agilent EESof products such as **ADS**.

Notes/Equations

1. This model reads data in Signal Studio file format either from a built-in waveform file (*WavefromFile*=**Built In**) or from a file specified by the *File* parameter (*WavefromFile*= **User Defined**).
2. The *SetCarrierFrequency* parameter controls the carrier frequency assigned to the output signal:
   - if set to *NO*, then the carrier frequency used is the one saved in the file header.
   - if set to *YES*, then the carrier frequency used is the one specified in the *RFCarrier* parameter.
3. The *Periodic* parameter controls what happens when the simulation requires more data than what is stored in the file:
   - if set to *NO*, then the additional data needed has a value of 0 + j·0.
   - if set to *YES*, then the data in the file is repeated.
4. The *SampleRateOption* parameter allows the user to select the sampling rate of the output signal:
   - if set to *Timed from source data*, then the sampling rate that is saved in the file header is used.
   - if set to *Timed from SampleRate*, then the sampling rate specified in the *SampleRate* parameter is used.
   - if set to *Timed from Schematic*, then the sampling rate is automatically determined based on the rest of the schematic (for more details see *Sampling Rate Resolution* (sim)).
5. If *SampleRateOption* is not set to *Timed from source data*, then interpolation is needed to get the signal values in between the actual signal samples that have been recorded at a different sampling rate. The *InterpolationType* parameter specifies the interpolation method that will be used. There are three available option: *Sample and hold*, *Linear*, and *Lagrange*. When *Lagrange* is selected the order of interpolation is specified in the *InterpolationOrder* parameter.
6. Understand ***how BurstMode works* (algorithm)** and ***how it interacts with Periodic mode* (algorithm)**.
   1. In a nutshell, you can think of *BurstMode* = **OFF** equivalent to: *BurstMode* = **Single**, *BurstLength* = *BurstPeriod* = "**Infinity**", and *BurstDelay* = 0.
   2. When *BurstMode* = **Multiple, each and every burst should have identical waveform signals**. The gap between *BurstPeriod* and *BurstLength* is padded with 0's.
   3. Within the duration of *BurstLength*, if *BurstLength* is longer than the duration of the waveform provided by the Signal Studio waveform file, the following will happen:
      1. If *Periodic* = **NO**, then once all the samples contained in the Signal Studio waveform file have been read out, 0's are padded for either direct output (when *SampleRateOption* = **Timed from source data**) or for interpolation (when *SampleRateOption* = **Timed from SampleRate** or **Timed from Schematic**).
      2. If *Periodic* = **YES**, then the samples contained in the Signal Studio waveform file will be repeatly read out for either direct output (when *SampleRateOption* = **Timed from source data**) or for interpolation (when *SampleRateOption* = **Timed from SampleRate** or **Timed from Schematic**)

See:
ReadFile (algorithm)
ReadN5106AFile (algorithm)
ReadN6030File (algorithm)
ReadBaseBandStudioFile (algorithm)
Sink (algorithm)

Important Links

1. Learn more about Signal Studio SW from Agilent Technologies.
2. Also see the *Sink* (algorithm) part on how to write modulated RF simulation data into a Signal Studio file.

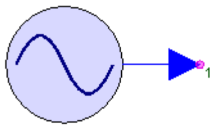
SineGen Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SineGen (algorithm)	Sine Wave Output

SineGen (Sine Wave Output)



Description: Sine Wave Output
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *SineGen Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
Amplitude	Peak sine wave voltage amplitude	1	V	Float	YES	(-∞:∞)	A
Offset	DC offset voltage	0	V	Float	YES	(-∞:∞)	DC
Frequency	Frequency	5e3	Hz	Float	YES	[0:SampleRate/4]	F
Phase	Phase	0	deg	Float	YES	(-∞:∞)	PH
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		B
BurstLength	Burst time length	100e-6	s	Float	YES	[1/SampleRate:∞)	BL
BurstPeriod	Time period from start of one burst to start of next	200e-6	s	Float	YES	[1/SampleRate:∞)	BP
BurstDelay	Time delay within burst before the start of the BurstLength interval	0	s	Float	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO

Notes/Equations

1. The function, A sin(2 π (F $t_{function}$ + PH / 360)) + DC, is output. For $t_{function}$ definition, see *Time Burst Sources* (algorithm).

2. For other parameter descriptions, see *Time Burst Sources* (algorithm).
3. Fill value is 0.
4. Quiescent value is DC.

See:
Oscillator Nonlinear Gain
ComplexExpGen (algorithm)
SineSweepGen (algorithm)Oscillator Nonlinear Gain

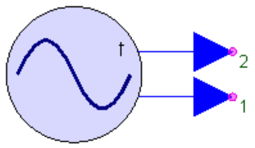
SineSweepGen Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SineSweepGen (algorithm)	Sine Wave Generator with Frequency Sweep

SineSweepGen (Sine Wave Generator with Frequency Sweep)



Description: Sine Wave Generator with Frequency Sweep
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *SineSweepGen Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
Amplitude	Peak sine wave voltage amplitude	1	V	Float	YES	(-∞:∞)	A
Offset	DC offset voltage	0	V	Float	YES	(-∞:∞)	DC
FSweepType	Frequency sweep type: linear, log	linear		Enumeration	YES		FT
StartFreq	Start frequency	1e3	Hz	Float	YES	(0:SampleRate/4]	F1
StopFreq	Stop frequency	10e3	Hz	Float	YES	(0:SampleRate/4]	F2
Phase	Phase	0	deg	Float	YES	(-∞:∞)	PH
SweepPeriod	Time period to complete a sweep	Stop_Time	s	Float	YES	[1/SampleRate:∞)	SP
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO
2	frequency	real	NO

Notes/Equations

1. The SineSweepGen model outputs a sine wave with *Amplitude* , *Phase* and *Offset* and swept in the frequency domain from *StartFreq* to 2**StopFreq* over the time interval *SweepPeriod*.
2. Let F(t $_{sweep}$) be the instantaneous time domain sweep frequency defined in *Time Swept Sources* (algorithm).
3. The waveform value, A sin(2 π (F(t $_{sweep}$) t $_{sweep}$ + PH / 360)) + DC, is output.
4. F(t $_{sweep}$) is output at the frequency port in Hertz. The maximum frequency domain spectral frequency is obtained when the time domain F(t $_{sweep}$) value is at its

maximum value of (*StopFreq* + *StartFreq*).

5. For other parameter descriptions, see *Time Swept Sources* (algorithm).
6. Fill value is 0.

See:
ChirpGen (algorithm)
ComplexExpGen (algorithm)
SineGen (algorithm)

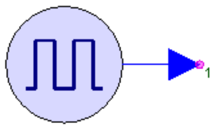
SquareGen Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SquareGen (algorithm)	Square Wave Generator

SquareGen (Square Wave Generator)



Description: Square Wave Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *SquareGen Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
LoLevel	Low voltage level	0	V	Float	YES	(-∞:∞)	VL
HiLevel	High voltage level	1	V	Float	YES	(LoLevel:∞)	VH
Frequency	Waveform frequency	5e3	Hz	Float	YES	[0:SampleRate/4]	F
Phase	Phase	0	deg	Float	YES	(-∞:∞)	PH
DutyCycle	Duty cycle in percent	50		Float	YES	[100*Frequency/SampleRate:100*(1-Frequency/SampleRate)]	DC
Polarity	Signal polarity: normal, inverted	normal		Enumeration	YES		P
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		B
BurstLength	Burst time length	100e-6	s	Float	YES	[1/SampleRate:∞)	BL
BurstPeriod	Time period from start of one burst to start of next	200e-6	s	Float	YES	[1/SampleRate:∞)	BP
BurstDelay	Time delay within burst before the start of the BurstLength interval	0	s	Float	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO

Notes/Equations

1. A square wave W(t) is defined on the t from [0, 1 / F).
2. Let t_{edge} be the terminating edge of the square wave at DC / 100 / F.

3. The output W(t) for normal Polarity is:
    - VH, for $0 \le t < t_{edge}$
    - VL, for $t_{edge} \le t < 1 / F$
4. For inverted Polarity, the square wave begins at VL and goes to VH.
5. For simulation time $t_{function}$, t is ($t_{function}$ + PH' / 360 / F) modulo (1 / F), where PH' is ((PH modulo 360) + 360) modulo 360.
   If zero PH, t is $t_{function}$ modulo (1 / F). For $t_{function}$ definition, see *Time Burst Sources* (algorithm).
6. For other parameter descriptions, see *Time Burst Sources* (algorithm).
7. Fill value is 0.
8. The quiescent value is:
    - VL, if Polarity (P) is normal
    - VH, otherwise

See:
SquareSweepGen (algorithm)
Impulse (algorithm)
PulseGen (algorithm)

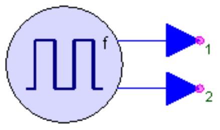
SquareSweepGen Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
SquareSweepGen (algorithm)	Square Sweep Wave Generator

SquareSweepGen (Square Sweep Wave Generator)



Description: Square Sweep Wave Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *SquareSweepGen Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
LoLevel	Low voltage level	0	V	Float	YES	(-∞:∞)	VL
HiLevel	High voltage level	1	V	Float	YES	(LoLevel:∞)	VH
FSweepType	Frequency sweep type: linear, log	linear		Enumeration	YES		FT
StartFreq	Start frequency	1e3	Hz	Float	YES	(0:SampleRate/4]	F1
StopFreq	Stop frequency	10e3	Hz	Float	YES	(0:SampleRate/4]	F2
Phase	Phase	0	deg	Float	YES	(-∞:∞)	PH
SweepPeriod	Time period to complete a sweep	Stop_Time	s	Float	YES	[1/SampleRate:∞)	SP
DutyCycle	Duty cycle in percent	50		Float	YES	[100*max(StartFreq,StopFreq)/SampleRate:100*(1-max(StartFreq,StopFreq)/SampleRate)]	DC
Polarity	Signal polarity: normal, inverted	normal		Enumeration	YES		P
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Initial output time delay	0	s	Float	YES	[0:∞)	D

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO
2	frequency	real	NO

Notes/Equations

1. Let F(t $_{sweep}$) be the instantaneous sweep frequency defined in *Time Swept Sources* (algorithm).
2. A square wave function W(t') is defined on t' from [0, 1 / F(t $_{sweep}$)). Let t $_{edge}$ be DC / 100 / F(t $_{sweep}$) be the terminating edge of the square wave.
3. Let t be (t $_{sweep}$ + (PH' / 360 / F(t $_{sweep}$))) modulo (1 / F(t $_{sweep}$)), where PH' = ((PH

modulo 360) + 360) modulo 360.
If PH is zero, t is t_{sweep} modulo (1 / F(t_{sweep})).

4. For *normal* Polarity, the output W(t) is:
   VH, if $0 \le t < t_{edge}$

   VL, if $t_{edge} \le t < 1 / F(t_{sweep})$
5. For *inverted* Polarity, the output W(t) is:
   VL, for $0 \le t < t_{edge}$

   VH, for $t_{edge} \le t < 1 / F(t_{sweep})$
6. F(t_{sweep}) is output at the frequency port in Hertz.
7. For other parameter descriptions, see *Time Swept Sources* (algorithm).
8. Fill value is 0.

See:
SquareGen (algorithm)
Impulse (algorithm)
PulseGen (algorithm)

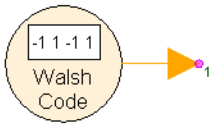
WalshCode Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
WalshCode (algorithm)	Walsh Code Generator

WalshCode (Walsh Code Generator)



Description: Walsh Code Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *WalshCode Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
WalshCodeType	Walsh code type: Walsh, Hadamard, OVSF_3GPP	Walsh		Enumeration	YES		T
Length	Code length (power of 2)	8		Integer	YES	[1:8192] †	L
Index	Code index	0		Integer	YES	[0:Length - 1]	I
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D

Output Ports

Port	Name	Signal Type	Optional
1	output	int	NO

Notes/Equations

1. One of three Walsh spreading codes are output.
2. For Length (L), spreading codes specified with different Index (I) values are orthogonal.
3. After L bits (chips), the code is reused.
4. The Index parameter is N in the *Walsh* description or selects a matrix row in the *Hadamard* or *OVSF_3GPP* description.
5. The output (chip) has index K in the *Walsh* description or is the K-th element of the matrix row in the *Hadamard* or *OVSF_3GPP* description.
6. For *Walsh*, the walsh codes are defined as:

$$h_{NK} = (-1)^{\sum_{i=0} r_i(n) k_i}$$

where
N is the index of the walsh code, [0, Length-1]
$N = n_{J-1}\ n_{J-2} \ldots n_1\ n_0$

K is the index of the chip in a walsh code, [0, Length-1]
$K = k_{J-1}\ k_{J-2} \ldots k_1\ k_0$

$J = \log_2(\text{Length})$

$r_0(n) = n_{J-1}$

$r_1(n) = n_{J-1} + n_{J-2}$

$r_2(n) = n_{J-2} + n_{J-3}$

.

.
.
$r_{J-1}(n) = n_1 + n_0$

7. For *Hadamard*, the walsh codes are defined as:
$$[H_2] = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$
$$[H_4] = \begin{bmatrix} H_2 & H_2 \\ H_2 & -H_2 \end{bmatrix}$$
.
.
.
$$[H_{2^m}] = \begin{bmatrix} H_{2^m} & H_{2^m} \\ H_{2^m} & -H_{2^m} \end{bmatrix}$$
8. For *OVSF_3GPP*, the walsh codes are defined as:
$C_0(0) = 1$
$$\begin{bmatrix} C_1(0) \\ C_1(1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$
$$\begin{bmatrix} C_{n+1}(0) \\ C_{n+1}(1) \\ C_{n+1}(2) \\ C_{n+1}(3) \\ C_{n+1}(2^{n+1}-2) \\ C_{n+1}(2^{n+1}-1) \end{bmatrix} = \begin{bmatrix} C_n(0) & C_n(0) \\ C_n(0) & -C_n(0) \\ C_n(1) & C_n(1) \\ C_n(1) & -C_n(1) \\ C_n(2^n-1) & C_n(2^n-1) \\ C_n(2^n-1) & \overline{C_n(2^n-1)} \end{bmatrix}$$
9. For other parameter descriptions, see *Untimed Sources* (algorithm).
10. Fill value is 0.

See:
RandomBits (algorithm)
PRBS (algorithm)
DataPattern (algorithm)

References

1. 3GPP Technical Specification 25.213 V3.0.0 "Spreading and modulation (FDD)", October 1999

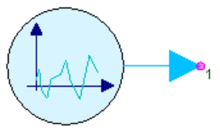
WaveForm Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	**Description**
WaveForm (algorithm)	Arbitrary Waveform

WaveForm (Arbitrary Waveform)



Description: Arbitrary Waveform
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *WaveForm Part* (algorithm)

Model Parameters

Name	**Description**	**Default**	**Units**	**Type**	**Runtime Tunable**	**Range**	**Symbol**
ExplicitValues	Waveform values	[1; 1; 1; 0; 0; 0]	V	None	NO		V
Offset	DC offset voltage	0	V	None	YES		DC
Periodic	Periodic signal: NO, YES	YES		Enumeration	YES		P
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D
BurstMode	Burst mode: OFF, Single, Multiple	OFF		Enumeration	YES		
BurstLength	Burst sample length	100		Integer	YES	[1:∞)	BL
BurstPeriod	Samples from start of one burst to start of next	200		Integer	YES	[1:∞)	BP
BurstDelay	Sample delay within burst before the start of the burst length interval	0		Integer	YES	[0:BurstPeriod-BurstLength]	DB

Output Ports

Port	**Name**	**Signal Type**	**Optional**
1	output	anytype	NO

Notes/Equations

1. A waveform specified by ExplicitValues biased by DC is output. For discussion on the variant type, see *Variant* (sim).
2. ExplicitValues is best defined with Equations. Equations can be localized to a schematic by creating an Equations tab in the design for the schematic.
   - Create a waveform of scalars by defining a variable that is a column vector, e.g. the statement, u = [0; 1; 2; 3; 4], creates a [5 x 1] matrix and defines a ramp of 5 samples. Note that statement, v = [1 2 3 4 5], creates a row vector which

is a [1 x 5] matrix and defines a waveform that is row vector of 1 sample.

- Create a waveform of vectors by defining a variable that is a matrix, e.g. the statement, u =[0 0; 1 2; 2 4; 3 6; 4 8] creates a [5 x 2] matrix and defines a vector of 2 ramp waveforms of 5 samples long.
- Create a waveform of matrices by defining a variable with the first matrix value and appending additional matrices with an additional right index. When finished, use the permute function to rotate the additional right index to the row index position. For example, the following steps creates a [5 x 2 x 2] matrix and defines a matrix of 4 ramp waveforms of 5 samples long.
  1. Define u as a [2 x 2] zero matrix with the statement, x = zeros(2, 2).
  2. Add the next 4 samples to u with the compound statement, u(:,:,2) = [1 2; 3 4]; u(:,:,2) = [2 4; 6 8]; ... ; u(:,:,5) = [4 8; 12 16].
  3. Permute the indices of u with the statement, u = permute(u, [3 1 2]).
- Note that waveforms of matrices with dimensionality greater than 2 can be constructed using the previous technique.

3. s_{function} = 0 always refer to first waveform value, and so on. For s_{function} definition, see *Untimed Burst Sources* (algorithm).
4. If the Periodic parameter is *YES*, the waveform may be repeated, otherwise the waveform is followed by DC.
5. For other parameter descriptions, see *Untimed Burst Sources* (algorithm).
6. Fill value is 0.
7. Quiescent value is the previous output.

See:
ReadFile (algorithm)
Window (algorithm)

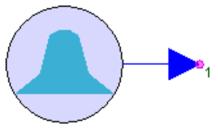
Window Part

Categories: *Sources* (algorithm)

The models associated with this part are listed below. To view detailed information on a model (description, parameters, equations, notes, etc.), please click the appropriate link.

Model	Description
Window (algorithm)	Window Generator

Window (Window Generator)



Description: Window Generator
Domain: Timed
C++ Code Generation Support: NO
Associated Parts: *Window Part* (algorithm)

Model Parameters

Name	Description	Default	Units	Type	Runtime Tunable	Range	Symbol
WindowType	Window type: RECTANGLE, BARTLETT, HANNING, HAMMING, BLACKMAN, STEEPBLACKMAN, KAISER	HANNING		Enumeration	YES		T
Length	Window sample length	256		Integer	YES	[4:∞)	L
ZeroPad	Number of zero values appended to length	0		Integer	YES	[0:∞)	Z
KaiserParameter	Beta parameter	0		Float	YES		Beta
ShowAdvancedParams	Show advanced parameters: NO, YES	NO		Enumeration	NO		
SampleRateOption	Sample rate option: UnTimed, Timed from SampleRate, Timed from Schematic	Timed from Schematic		Enumeration	NO		SO
SampleRate	Explicit sample rate	Sample_Rate	Hz	Float	NO	(0:∞)	S
InitialDelay	Output sample delay	0		Integer	YES	[0:∞)	D

Output Ports

Port	Name	Signal Type	Optional
1	output	real	NO

Notes/Equations

1. A periodic waveform consist of Length (L) samples of a window as selected by WindowType (T) followed by ZeroPad (Z) zero samples.
2. The window is centered around its maximum in L samples.
3. The KaiserParameter (Beta) value controls the stopband attenuation of the Kaiser window. A larger absolute value gives greater stopband attenuation.
4. For other parameter descriptions, see *Untimed Sources* (algorithm).
5. Fill value is 0.

See:
ReadFile (algorithm)
WaveForm (algorithm)

References

1. Leland Jackson, *Digital Filters and Signal Processing*, 2nd ed., Kluwer Academic Publishers, ISBN 0-89838-276-9, 1989.