Modeling Guidelines for Code Generation

R2011b

MATLAB®
&SIMULINK®

MathWorks®

Revision History

September 2010	Online only	New for Version 1.0 (Release 2010b)
April 2011	Online only	Revised for Version 1.1 (Release 2011a)
September 2011	Online only	Revised for Version 1.2 (Release 2011b)

Contents

Introduction

Motivation

MathWorks intends this document for engineers developing models and generating code for embedded systems using Model-Based Design with MathWorks® products. The document focus is on model settings, block usage, and block parameters that impact simulation behavior or code generation.

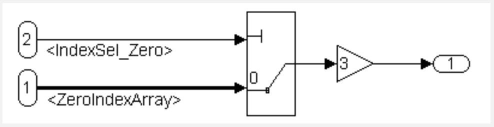
This document does not address model style or development processes. For more information about creating models in a way that improves consistency, clarity, and readability, see the "MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB®, Simulink®, and Stateflow®". Development process guidance and additional information for specific standards is available with the IEC Certification Kit (for IEC 61508 and ISO 26262) and DO Qualification Kit (for DO-178B) products.

Disclaimer While adhering to the recommendations in this document will reduce the risk that an error is introduced during development and not be detected, it is not a guarantee that the system being developed will be safe. Conversely, if some of the recommendations in this document are not followed, it does not mean that the system being developed will be unsafe.

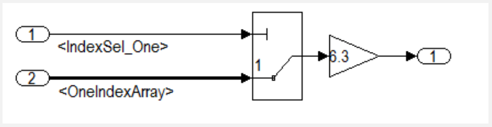
Block Considerations

cgsl_0101: Zero-based indexing

ID: Title	cgsl_0101: Zero-based indexing	
Description	Use zero-based indexing for blocks that require indexing. To set up zero-based indexing, do one of the following:	
	A	Select block parameter **Use zero-based indexing** for the Index Vector block.
	B	Set block parameter **Index mode** to `Zero-based` for the following blocks: • Assignment • Selector • For Iterator
Notes	The C language uses zero-based indexing.	
Rationale	A, B	Use zero-based indexing for compatibility with integrated C code.
	A, B	Results in more efficient C code execution. One-based indexing requires a subtraction operation in generated code.
See Also	"hisl_0021: Consistent vector indexing method"	
Last Changed	R2011b	

Examples	
	Recommended
	`void ZeroIndex(void)`
	`{`
	`  Y.Out5 = 3.0 * ZeroIndexArray[IndexSel_Zero];`
	`}`	

ID: Title	cgsl_0101: Zero-based indexing

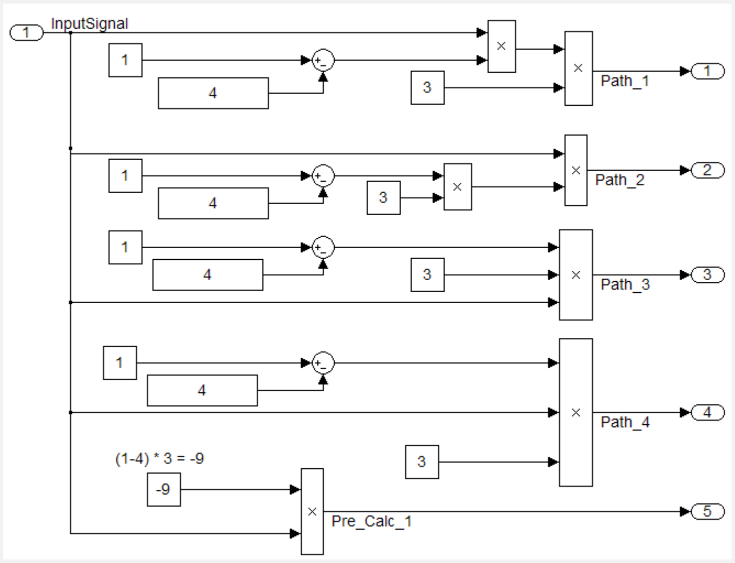
	 **Not Recommended**
	`void OneIndex(void)`
	`{`
	`  Y.Out1 = OneIndexArray[IndexSel_One - 1] * 6.3;`
	`}`

cgsl_0102: Evenly spaced breakpoints in lookup tables

ID: Title	**cgsl_0102: Evenly spaced breakpoints in lookup tables**	
Description	When you use Lookup Table and Prelookup blocks,	
	A	With *non-fixed-point data types*, use evenly spaced data breakpoints for the input axis
	B	With *fixed-point data types*, use power of two spaced breakpoints for the input axis
Notes	Evenly-spaced breakpoints can prevent generated code from including division operations, resulting in faster execution.	
Rationale	A	Improve ROM usage and execution speed.
	B	Improve execution speed. When compared to unevenly-spaced data, power-of-two data can • Increase data RAM usage if you require a finer step size • Reduce accuracy if you use a coarser step size Compared to an evenly-spaced data set, there should be minimal cost in memory or accuracy.
Model Advisor Checks	**Embedded Coder > "Identify questionable fixed-point operations"**	
See Also	"Formulation of Evenly Spaced Breakpoints" in the Simulink documentation	
Last Changed	R2010b	

cgsl_0103: Precalculated signals and parameters

ID: Title	**cgsl_0103: Precalculated signals and parameters**	
Description	Precalculate invariant parameters and signals by doing one of the following:	
	A	Manually precalculate the values
	B	Enable the following model optimization parameters: • **Optimization > Simulation and code generation > Inline parameters** • **Optimization > Code generation > Signals > Inline invariant signals**
Notes	Precalculating variables can reduce local and global memory usage and improve execution speed. If you select **Inline parameters** and **Inline invariant signals**, the code generator minimizes the number of run-time calculations by maximizing the number calculations completed before runtime. In some cases, this can lead to a reduction in the number of parameters stored. However, the algorithms the code generator uses have limitations. In some cases, the code is more compact if you calculate the values outside of the Simulink environment. This can improve model efficiency, but can reduce model readability.	
Rationale	A, B	Precalculate data, outside of the Simulink environment, to reduce memory requirements of a system and improve run-time execution.
Last Changed	R2010b	
Examples	In the following model, all four paths are mathematically equivalent. However, due to algorithm limitations, the number of run-time calculations for the paths differs.	

ID: Title	cgsl_0103: Precalculated signals and parameters
	 InputSignal Path_1 Path_2 Path_3 Path_4 (1-4) * 3 = -9 Pre_Calc_1

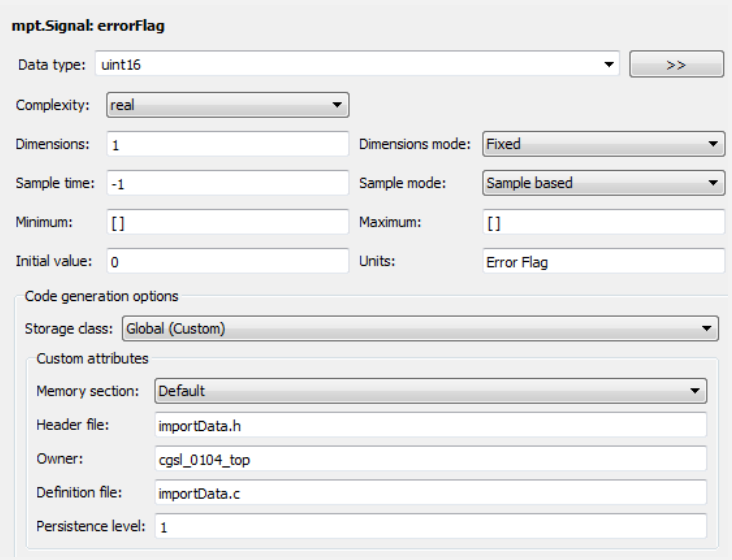
```
Path_1 = InputSignal * -3.0 * 3.0;

/* Product: '<Root>/Product4' incorporates:
 *  Inport: '<Root>/In1'
 */
Path_2 = InputSignal * -9.0;

/* Product: '<Root>/Product2' incorporates:
 *  Constant: '<Root>/Constant2'
 *  Inport: '<Root>/In1'
 */
Path_3 = -9.0 * InputSignal;

/* Product: '<Root>/Product5' incorporates:
 *  Constant: '<Root>/Constant2'
 *  Inport: '<Root>/In1'
 */
Path_4 = -3.0 * InputSignal * 3.0;
```

ID: Title	cgsl_0103: Precalculated signals and parameters

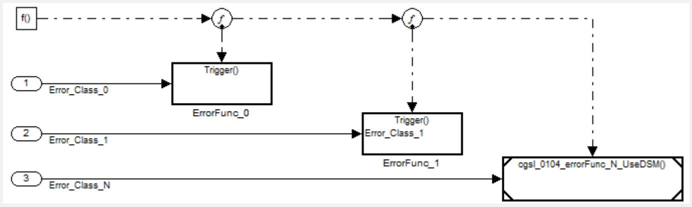
	`/* Product: '<Root>/Product6' incorporates:`
	` *  Constant: '<Root>/Constant3'`
	` *  Inport: '<Root>/In1'`
	` */`
	`Pre_Calc_1 = -9.0 * InputSignal;`
	
	To maximize automatic precalculation, add signals at the end of the set of equations.
	
	Inlining data reduces the ability to tune model parameters. You should define parameters that require calibration to allow calibration. For more information, see “Parameters” in the Simulink® Coder™ documentation.

cgsl_0104: Modeling global shared memory using data stores

ID: Title	**cgsl_0104: Modeling global shared memory using data stores**	
Description	When using data store blocks to model shared memory across multiple models:	
	A	In the Configuration Parameters dialog box, on the **Diagnostics** pane, set **Data Validity > Data Store Memory Block > Duplicate data store names** to `error` for **all** the models in the hierarchy
	B	Define the data store using a Simulink Signal or MPT Signal object
	C	Do not use Data Store Memory blocks in any of the models
Notes	If multiple Data Store blocks use the same data store name within a model, then Simulink interprets each instance of the data store as having a unique local scope. Use the diagnostic **Duplicate data store names** to help detect unintended identifier reuse. For models intentionally using local data stores, set the diagnostic to `warning`. Verify that only intentional data stores are included. Merge blocks, used in conjunction with subsystems operating in a mutually exclusive manor, provide a second method of modeling global data across multiple models.	
Rationale	A, B, C	Promotes a modeling pattern where a single consistent data store is used across all models and a single global instance is created in the generated code.

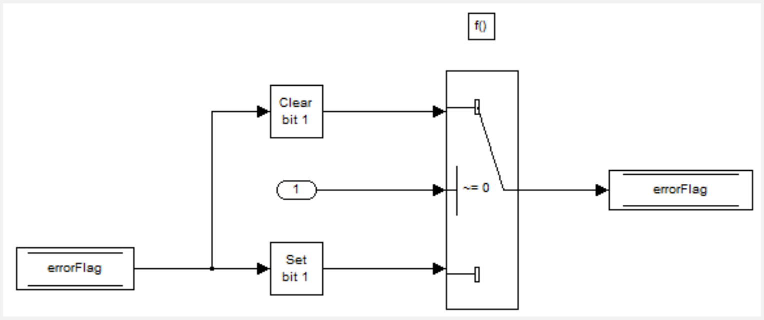
ID: Title	**cgsl_0104: Modeling global shared memory using data stores**
See Also	• "hisl_0013: Usage of data store blocks" • "hisl_0015: Usage of Merge blocks" • "cgsl_0302: Diagnostic settings for multirate and multitasking models" • "cgsl_0105: Modeling local shared memory using data stores"
Last Changed	R2011b
Examples	The following examples illustrate the use of data stores as global shared memory. The data store is used to model a global fault flag. A data store is required because the flag can be set in multiple functions and used in the same execution step. The top model contains three subsystems, each utilizing a data store memory. The data store is defined using a `mpt.Signal` object.  mpt.Signal: errorFlag Data type: uint16 Complexity: real Dimensions: 1 Dimensions mode: Fixed Sample time: -1 Sample mode: Sample based Minimum: [] Maximum: [] Initial value: 0 Units: Error Flag Code generation options Storage class: Global (Custom) Custom attributes Memory section: Default Header file: importData.h Owner: cgsl_0104_top Definition file: importData.c Persistence level: 1

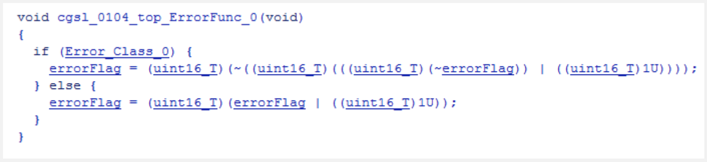
ID: Title	cgsl_0104: Modeling global shared memory using data stores



Recommended

In this example, there are no Data Store Memory blocks. The resulting code uses the same global variable for the full model.

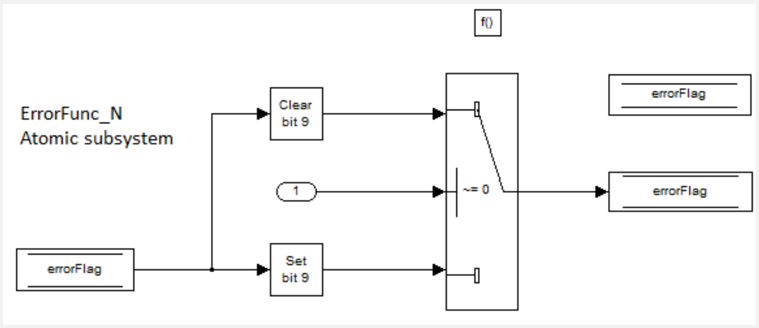
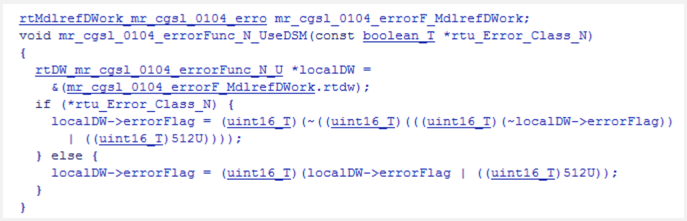




```
void cgsl_0104_top_ErrorFunc_0(void)
{
  if (Error_Class_0) {
    errorFlag = (uint16_T)(~((uint16_T)(((uint16_T)(~errorFlag)) | ((uint16_T)1U))));
  } else {
    errorFlag = (uint16_T)(errorFlag | ((uint16_T)1U));
  }
}
```

Not Recommended

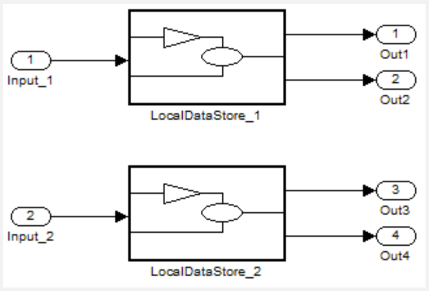
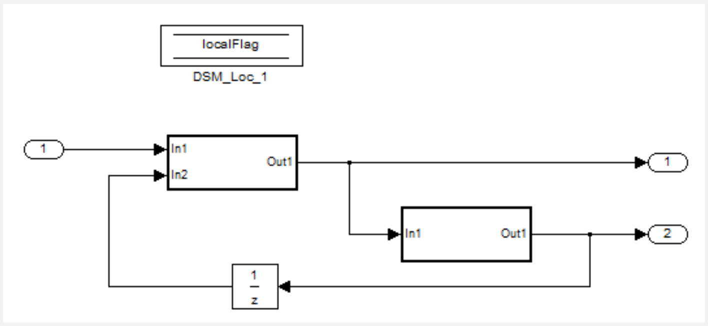
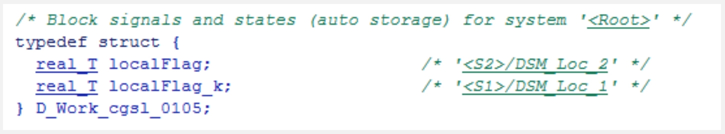
In this example, a Data Store Memory block is added into the Model block subsystem. The model subsystem uses a local version of the data store. The Atomic Subsystem use a different version.

ID: Title	**cgsl_0104: Modeling global shared memory using data stores**
	 f() ErrorFunc_N Atomic subsystem Clear bit 9 1 ~= 0 errorFlag errorFlag errorFlag Set bit 9 

```
rtMdlrefDWork_mr_cgsl_0104_erro mr_cgsl_0104_errorF_MdlrefDWork;
void mr_cgsl_0104_errorFunc_N_UseDSM(const boolean_T *rtu_Error_Class_N)
{
  rtDW_mr_cgsl_0104_errorFunc_N_U *localDW =
    &(mr_cgsl_0104_errorF_MdlrefDWork.rtdw);
  if (*rtu_Error_Class_N) {
    localDW->errorFlag = (uint16_T)(~((uint16_T)(((uint16_T)(~localDW->errorFlag))
      | ((uint16_T)512U))));
  } else {
    localDW->errorFlag = (uint16_T)(localDW->errorFlag | ((uint16_T)512U));
  }
}
```

cgsl_0105: Modeling local shared memory using data stores

ID: Title	**cgsl_0105: Modeling local shared memory using data stores**	
Description	When using data store blocks as local shared memory:	
	A	Explicitly create the data store using a Data Store Memory block.
	B	Deselect the block parameter option **Data store name must resolve to Simulink signal object**.
	C	Consider following a naming convention for local Data Store Memory blocks.
Notes	Use the diagnostic **Duplicate data store names** to help detect unintended identifier reuse. For models intentionally using local data stores, set the diagnostic to `warning`. Verify that only intentional data stores are included. Data store blocks are realized as global memory in the generated code. If they are not assigned a specific storage class, they are included in the DWork structure. In the model, the data store is scoped to the defining subsystem and below. In the generated code, the data store has file scope.	
Rationale	A, B	Data store block is treated as a local instance of the data store
	C	Provides graphical feedback that the data store is local
See Also	• "cgsl_0104: Modeling global shared memory using data stores" • "cgsl_0302: Diagnostic settings for multirate and multitasking models" • "hisl_0013: Usage of data store blocks"	

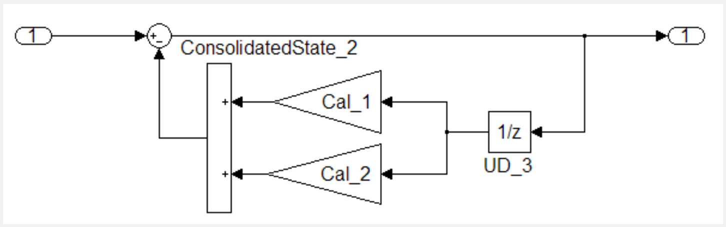
ID: Title	**cgsl_0105: Modeling local shared memory using data stores**
Last Changed	R2011b
Examples	In some instances, such as a library function, reuse of a local data store is required. In this example the local data store is defined in two subsystems.  1 Input_1 LocalDataStore_1 1 Out1 2 Out2 2 Input_2 LocalDataStore_2 3 Out3 4 Out4  localFlag DSM_Loc_1 1 In1 In2 Out1 In1 Out1 1 2 1/z The instance of `localFlag` is in scope within the subsystem `LocalDataStore_1` and its subsystems. 

```
/* Block signals and states (auto storage) for system '<Root>' */
typedef struct {
  real_T localFlag;                    /* '<S2>/DSM_Loc_2' */
  real_T localFlag_k;                  /* '<S1>/DSM_Loc_1' */
} D_Work_cgsl_0105;
```

ID: Title	cgsl_0105: Modeling local shared memory using data stores
	In the generated code, the data stores are part of the global DWork structure for the model. Embedded coder automatically assigns them unique names during the code generation process.

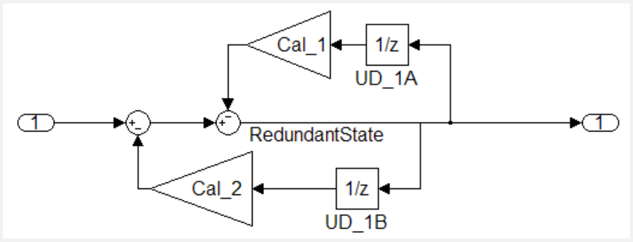
Modeling Pattern Considerations

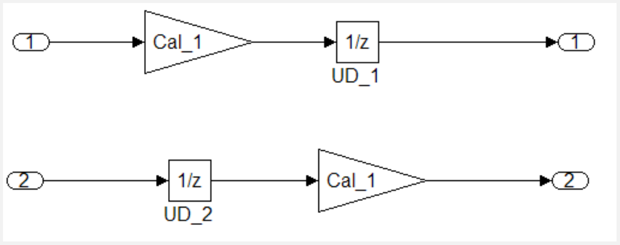
cgsl_0201: Eliminate redundant state blocks

ID: Title		cgsl_0201: Eliminate redundant state blocks
Description	When preparing a model for code generation,	
	A	Remove redundant Unit Delay and Memory blocks.
Rationale	A	Redundant Unit Delay and Memory blocks use additional global memory. Removing the redundancies from a model reduces memory usage without impacting model behavior.
Last Changed	R2010b	
Example	 1 ConsolidatedState_2 Cal_1 Cal_2 1/z UD_3 1 **Recommended: Consolidated Unit Delays** `void Reduced(void)` `{` `ConsolidatedState_2 = Matrix_UD_Test - (Cal_1 * DWork.UD_3_DSTATE + Cal_2 * DWork.UD_3_DSTATE);` `DWork.UD_3_DSTATE = ConsolidatedState_2;` `}`	

```
void Reduced(void)
{
  ConsolidatedState_2 = Matrix_UD_Test - (Cal_1 * DWork.UD_3_DSTATE + Cal_2 *
    DWork.UD_3_DSTATE);
  DWork.UD_3_DSTATE = ConsolidatedState_2;
}
```

ID: Title	cgsl_0201: Eliminate redundant state blocks

	 Cal_1
	1/z
	UD_1A
	1
	RedundantState
	1
	Cal_2
	1/z
	UD_1B **Not Recommended: Redundant Unit Delays**
	`void Redundent(void)`
	`{`
	`  RedundantState = (Matrix_UD_Test - Cal_2 * DWork.UD_1B_DSTATE) - Cal_1 *`
	`    DWork.UD_1A_DSTATE;`
	`  DWork.UD_1B_DSTATE = RedundantState;`
	`  DWork.UD_1A_DSTATE = RedundantState;`
	`}`

	Unit Delay and Memory blocks exhibit commutative and distributive algebraic properties. When the blocks are part of an equation with one driving signal, you can move the Unit Delay and Memory blocks to any position in the equation without changing the result.  1 Cal_1 1/z UD_1 1 2 1/z UD_2 Cal_1 2 For the top path in the preceding example, the equations for the blocks are: **1** `Out_1(t) = UD_1(t)` **2** `UD_1(t) = In_1(t-1) * Cal_1`

ID: Title	cgsl_0201: Eliminate redundant state blocks

3 `Out_1(t) = In_1(t-1) * Cal_1`

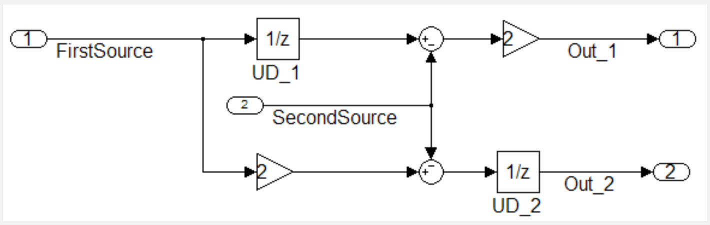
For the bottom path, the equations are:

1 `Out_2(t) = UD_2(t) * Cal_1`

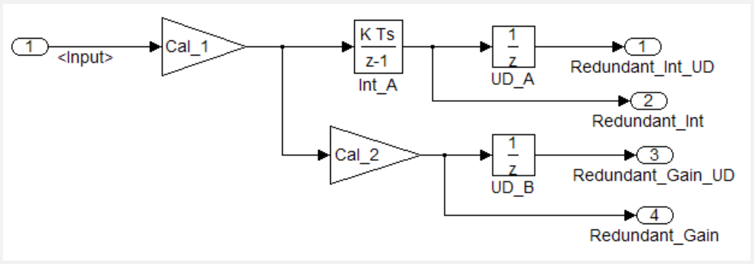
2 `UD_2(t) = In_2(t-1)`

3 `Out_2(t) = In_2(t-1) * Cal_1`

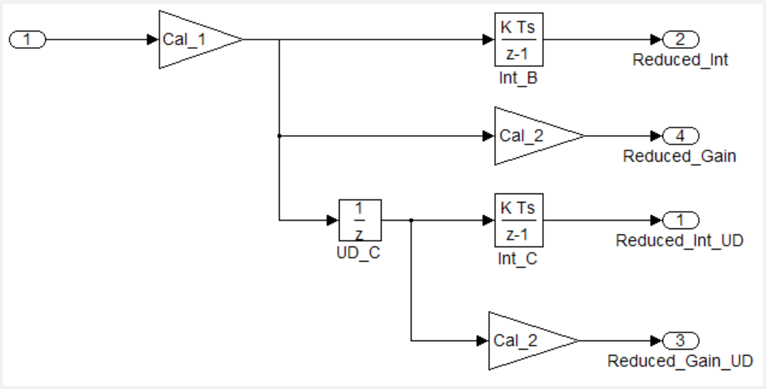
In contrast, if you add a secondary signal to the equations, the location of the Unit Delay block affects the result. As the following example shows, the location of the Unit Delay block affects the results due the skewing of the time sample between the top and bottom paths.



In cases with a single source and multiple destinations, the comparison is more complex. For example, in the following model, you can refactor the two Unit Delay blocks into a single unit delay.



ID: Title	cgsl_0201: Eliminate redundant state blocks



From a black box perspective, the two models are equivalent. However, from a memory and computation perspective, differences exist between the two models.

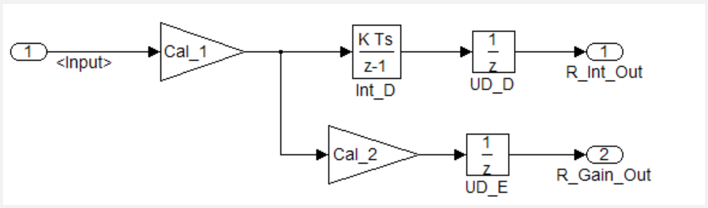
```
{
  real_T rtb_Gain4;
  rtb_Gain4 = Cal_1 * Redundant;
  Y.Redundant_Gain = Cal_2 * rtb_Gain4;
  Y.Redundant_Int = DWork.Int_A;
  Y.Redundant_Int_UD = DWork.UD_A;
  Y.Redundant_Gain_UD = DWork.UD_B;
  DWork.Int_A = 0.01 * rtb_Gain4 + DWork.Int_A;
  DWork.UD_A = Y.Redundant_Int;
  DWork.UD_B = Y.Redundant_Gain;
}

{
  real_T rtb_Gain1;
  real_T rtb_UD_C;
  rtb_Gain1 = Cal_1 * Reduced;
  rtb_UD_C = DWork.UD_C;
  Y.Reduced_Gain_UD = Cal_2 * DWork.UD_C;
  Y.Reduced_Gain = Cal_2 * rtb_Gain1;
  Y.Reduced_Int = DWork.Int_B;
  Y.Reduced_Int_UD = DWork.Int_C;
  DWork.UD_C = rtb_Gain1;
```

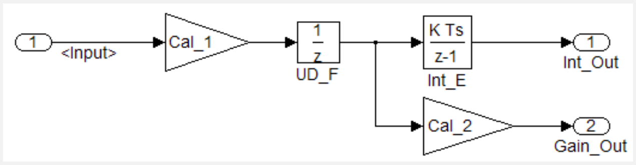
ID: Title	cgsl_0201: Eliminate redundant state blocks

```
  DWork.Int_B = 0.01 * rtb_Gain1 + DWork.Int_B;
  DWork.Int_C = 0.01 * rtb_UD_C + DWork.Int_C;
}

{
  real_T rtb_Gain4_f;
  real_T rtb_Int_D;
  rtb_Gain4_f = Cal_1 * U.Input;
  rtb_Int_D = DWork.Int_D;
  Y.R_Int_Out = DWork.UD_D;
  Y.R_Gain_Out = DWork.UD_E;
  DWork.Int_D = 0.01 * rtb_Gain4_f + DWork.Int_D;
  DWork.UD_D = rtb_Int_D;
  DWork.UD_E = Cal_2 * rtb_Gain4_f;
}
```

In this case, the original model is more efficient. In the first code example, there are three bits of global data, two from the Unit Delay blocks (DWork.UD_A and DWork.UD_B) and one from the discreate time integrator (DWork.Int_A). The second code example shows a reduction to one global variable generated by the unit delays (Dwork.UD_C), but there are two global variables due to the redundant Discreate Time Integrator blocks (DWork.Int_B and DWork.Int_C). The Discreate Time Integrator block path introduces an additional local variable (rtb_UD_C) and two additional computations.

By contrast, the refactored model (second) below is more efficient.



ID: Title	cgsl_0201: Eliminate redundant state blocks
	 1 <Input> Cal_1 1/z UD_F K Ts / z-1 Int_E 1 Int_Out Cal_2 2 Gain_Out

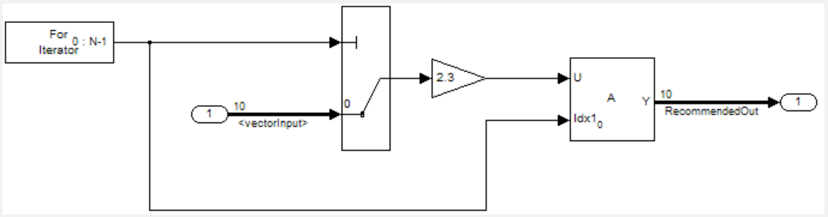
	`{`
	`  real_T rtb_Gain4_f:`
	`  real_T rtb_Int_D;`
	`  rtb_Gain4_f = Cal_1 * U.Input;`
	`  rtb_Int_D = DWork.Int_D;`
	`  Y.R_Int_Out = DWork.UD_D;`
	`  Y.R_Gain_Out = DWork.UD_E;`
	`  DWork.Int_D = 0.01 * rtb_Gain4_f + DWork.Int_D;`
	`  DWork.UD_D = rtb_Int_D;`
	`  DWork.UD_E = Cal_2 * rtb_Gain4_f;`
	`}`

	`{`
	`  real_T rtb_UD_F;`
	`  rtb_UD_F = DWork.UD_F;`
	`  Y.Gain_Out = Cal_2 * DWork.UD_F;`
	`  Y.Int_Out = DWork.Int_E;`
	`  DWork.UD_F = Cal_1 * U.Input;`
	`  DWork.Int_E = 0.01 * rtb_UD_F + DWork.Int_E;`
	`}`

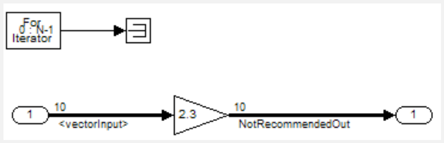
	The code for the refactored model is more efficient because no branches from the root signal have a redundant unit delay.

cgsl_0202: Usage of For, While, and For Each subsystems with vector signals

ID: Title	cgsl_0202: Usage of For, While, and For Each subsystems with vector signals	
Description	When developing a model for code generation,	
	A	Use For, While, and For Each subsystems for calculations that require iterative behavior or operate on a subset (frame) of data.
	B	Avoid using For, While, or For Each subsystems for basic vector operations.
Rationale	A, B	Avoid redundant loops.
See Also	• "Loop unrolling threshold" in the Simulink documentation • MathWorks Automotive Advisor Board guideline db_0117: Simulink patterns for vector signals	
Last Changed	R2010b	

Examples	The recommended method for preceding calculation is to place the Gain block outside the For Subsystem. If the calculations are required as part of a larger algorithm, you can avoid the nesting of `for` loops by using Index Vector and Assignment blocks.
	 For Iterator
	0 : N-1
	1
	10
	<vectorInput>
	0
	2.3
	U
	A
	Y
	Idx1
	0
	10
	RecommendedOut
	1
	Recommended
	`for (s1_iter = 0; s1_iter < 10; s1_iter++) {`
	`  RecommendedOut[s1_iter] = 2.3 * vectorInput[s1_iter];`
	`}`	

ID: Title	**cgsl_0202: Usage of For, While, and For Each subsystems with vector signals**
	A common mistake is to embed basic vector operations in a For, While, or For Each subsystem. The following example includes a simple vector gain inside a For subsystem, which results in unnecessary nested `for` loops.

	 For 0:N-1 Iterator
	1
	10
	<vectorInput>
	2.3
	10
	NotRecommendedOut
	1 **Not Recommended**
	`for (s1_iter = 0; s1_iter < 10; s1_iter++) {`
	`  for (i = 0; i < 10; i++) {`
	`    NotRecommendedOut[i] = 2.3 * vectorInput[i];`
	`  }`
	`}`

cgsl_0204: Vector and bus signals crossing into atomic subsystems

ID: Title		cgsl_0204: Vector and bus signals crossing into atomic subsystems
Description		When working with a bus or vector signal, where only part of the signal is used in an Atomic subsystem,
	A	Use the following tables applies to signals with **local** and **global** scope. It can be used to determine which parts of the signal to select to minimize memory usage: **Note** Virtual buses do not support global data.**Function**

	Signals selected outside subsystem results in...	**Signal selected inside subsystem results in...**
Virtual Bus	No data copies	No data copies
Non-Virtual Bus	A copy of all signals are placed in the global Block I/O structure	No data copies
Vector	No data copies	No data copies

Reusable Function

	Signals selected outside subsystem results in	**Signal selected inside subsystem results in**
Virtual Bus	No data copies, only the selected elements are passed into the function	No data copies, only the selected elements are passed into the function
Non-Virtual Bus	A copy of the full bus is placed into the global Block I/O structure, only the	No data copies; the full bus is passed in by reference.

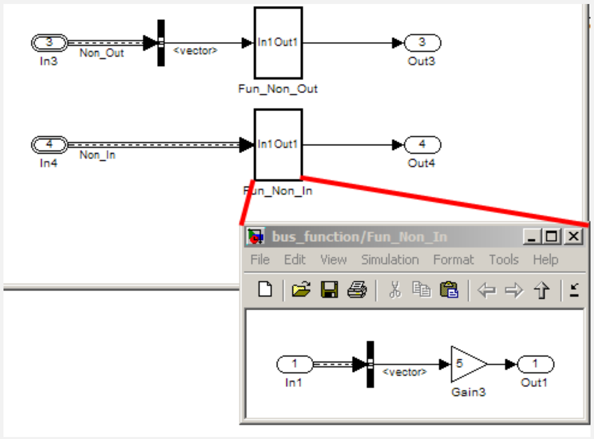
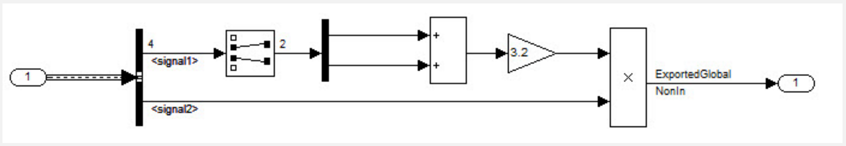
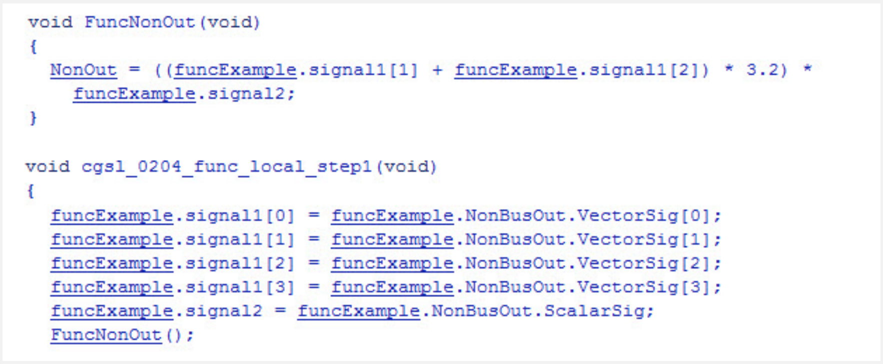
ID: Title		cgsl_0204: Vector and bus signals crossing into atomic subsystems
		(continued)
Rationale	A	Minimize ROM requirements.

	elements used in the function are passed.	
Vector	No data copies; only the vector elements used in the subsystem are passed into the function.	No data copies; only the vector elements used in the subsystem are passed into the function.

Model Reference

	Signals selected outside subsystem results in	**Signal selected inside the subsystem results in**
Virtual Bus	No data copies	Full bus copied; full bus passed into the function.
Non-Virtual Bus	Full bus copied; full bus passed into the function.	No data copies; full bus passed into the function
Vector	No data copies; selected only the vector elements used in the subsystem are passed into the function.	No data copies; full vector passed by reference

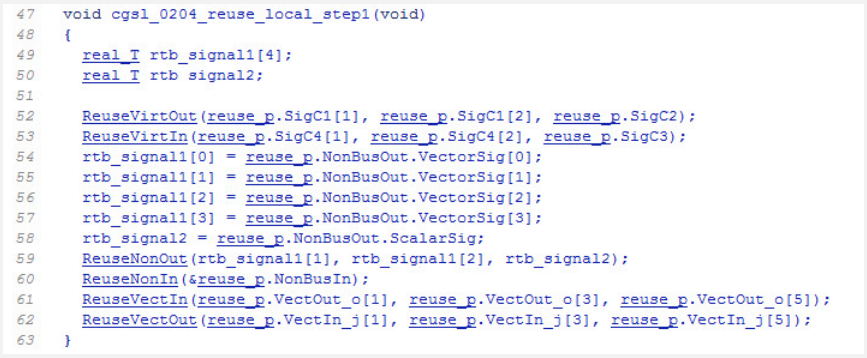
If the subsystem is set to `Inline`, no data copies occur.

ID: Title	cgsl_0204: Vector and bus signals crossing into atomic subsystems
Last Changed	R2011a
Examples	**Example of selecting signals inside and outside of an atomic subsystem**  In3 Non_Out <vector> In1 Out1 Fun_Non_Out Out3 In4 Non_In Fun_Non_In Out4 bus_function/Fun_Non_In File Edit View Simulation Format Tools Help In1 <vector> Gain3 Out1 **Signals selected inside the subsystem for a NonVirtual bus with the subsystem set to atomic and `Function`**  <signal1> <signal2> 3.2 ExportedGlobal NonIn 

```
void FuncNonOut(void)
{
  NonOut = ((funcExample.signal1[1] + funcExample.signal1[2]) * 3.2) *
    funcExample.signal2;
}

void cgsl_0204_func_local_step1(void)
{
  funcExample.signal1[0] = funcExample.NonBusOut.VectorSig[0];
  funcExample.signal1[1] = funcExample.NonBusOut.VectorSig[1];
  funcExample.signal1[2] = funcExample.NonBusOut.VectorSig[2];
  funcExample.signal1[3] = funcExample.NonBusOut.VectorSig[3];
  funcExample.signal2 = funcExample.NonBusOut.ScalarSig;
  FuncNonOut();
```

ID: Title	cgsl_0204: Vector and bus signals crossing into atomic subsystems

In this example the full bus is copied to the global variable *funcExample* even though only 3 of the signals from the bus are used. **Reusable function example**



```
47  void cgsl_0204_reuse_local_step1(void)
48  {
49    real_T rtb_signal1[4];
50    real_T rtb_signal2;
51
52    ReuseVirtOut(reuse_p.SigC1[1], reuse_p.SigC1[2], reuse_p.SigC2);
53    ReuseVirtIn(reuse_p.SigC4[1], reuse_p.SigC4[2], reuse_p.SigC3);
54    rtb_signal1[0] = reuse_p.NonBusOut.VectorSig[0];
55    rtb_signal1[1] = reuse_p.NonBusOut.VectorSig[1];
56    rtb_signal1[2] = reuse_p.NonBusOut.VectorSig[2];
57    rtb_signal1[3] = reuse_p.NonBusOut.VectorSig[3];
58    rtb_signal2 = reuse_p.NonBusOut.ScalarSig;
59    ReuseNonOut(rtb_signal1[1], rtb_signal1[2], rtb_signal2);
60    ReuseNonIn(&reuse_p.NonBusIn);
61    ReuseVectIn(reuse_p.VectOut_o[1], reuse_p.VectOut_o[3], reuse_p.VectOut_o[5]);
62    ReuseVectOut(reuse_p.VectIn_j[1], reuse_p.VectIn_j[3], reuse_p.VectIn_j[5]);
63  }
```

- Line 53 corresponds to a reusable function with a virtual bus selection inside of the atomic subsystem. Only the signals used by the function are passed into the function

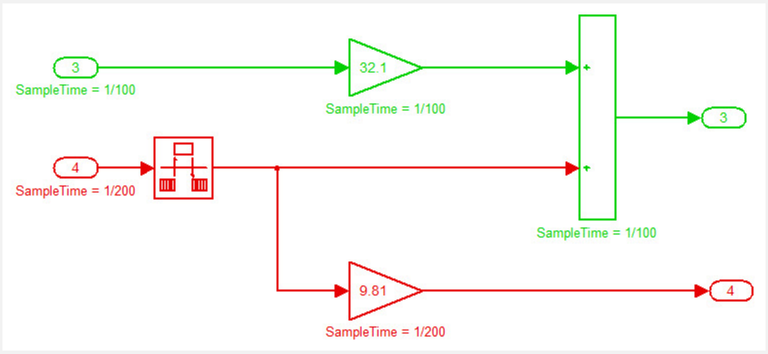
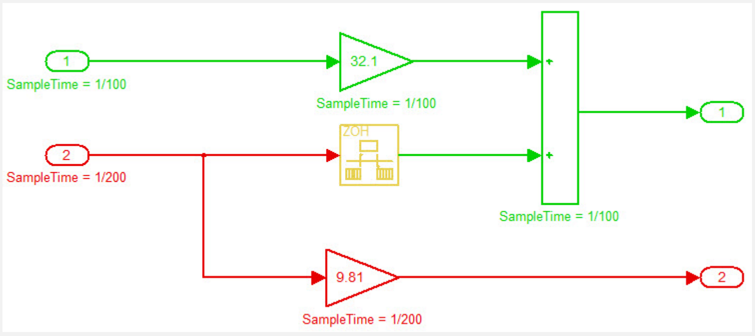
- Lines 54 through 59 show a nonvirtual bus with signals selected outside of the atomic subsystem. Copies of the data are placed into global storage *rtb_**, again only the data used by the function is passed

- Line 60 shows a nonvirtual bus with data selected inside of the atomic subsystem. The full bus is passed into the subsystem

- Line 61 shows the vector selected inside the atomic subsystem case. Only the signals used inside of the subsystem are passed into the function.

cgsl_0205: Signal handling for multirate models

ID: Title	**cgsl_0205: Signal handling for multirate models**	
Description	For multirate models, handle the change in operation rate in one of two ways:	
	A	At the destination block, Insert a Rate Transition.
	B	Set the parameter **Solver > Automatically handle rate transition for data transfer** to either `Always` or `Whenever possible`.
Rationale	A,B	Following this guideline ensures the proper handling of data operating at different rates.
Note	Setting the parameter **Solver > Automatically handle rate transition for data transfer with the setting** to `Whenever possible` requires inserting a Rate Transition block in locations indicated by Simulink. Setting the parameter **Solver > Automatically handle rate transition for data transfer** to `Always` allows Simulink to automatically handle all rate transitions by inserting a Rate Transition block. The following exceptions apply: • The insertion of a Rate Transition block requires rewiring the block diagram. • Multiple Rate Transition blocks are required: - The blocks' sample times are not integer multiples of each other - The blocks use different sample time offsets - One of the rates is asynchronous • An inserted Rate Transition block can have multiple valid configurations. For these cases, manually insert a Rate Transition block or blocks. MathWorks does not recommend using Unit Delay and Zero Order Hold blocks for handling rate transitions.	

ID: Title	cgsl_0205: Signal handling for multirate models
Last Changed	R2011a
Examples	**Incorrect:** In this example, the Rate Transition block is inserted at the source, not at the destination of the signal. The model fails to update because the two destination blocks (Gain and Sum) run at different rates. To fix this error, insert Rate Transition blocks at the signal destinations and remove Rate Transition blocks from the signal sources. Failure to remove the Rate Transition blocks is a common modeling pattern that might result in errors and inefficient code.  3 SampleTime = 1/100 32.1 SampleTime = 1/100 + + 3 4 SampleTime = 1/200 SampleTime = 1/100 9.81 4 SampleTime = 1/200 **Correct:** In this example, the rate transition is inserted at the destination of the signal.  1 SampleTime = 1/100 32.1 SampleTime = 1/100 + 1 ZOH 2 SampleTime = 1/200 + SampleTime = 1/100 9.81 2 SampleTime = 1/200

cgsl_0206: Data integrity and determinism in multitasking models

ID: Title		cgsl_0206: Data integrity and determinism in multitasking models
Description	For multitasking models that are deployed with a preemptive (interruptible) operating system, protect the **integrity** of selected signals by doing one of the following:	
	A	Select the Rate Transition block parameter **Ensure data integrity during data transfer**.
	B	For Inport blocks in Function Called subsystems, select the block parameter **Latch input for feedback signals of function-call subsystem outputs**.
	To protect selected signal **determinism,** do one of the following:	
	C	Select the Rate Transition block parameter **Ensure deterministic data transfer (maximum delay)**.
	D	• Select the model parameter **Solver > Automatically handle rate transition for data transfer**. • Set the model parameter **Solver > Deterministic data transfer** to either `Whenever possible` or `Always`.
Rationale	A,B, C,D	Following this guideline protects data against possible corruption of preemptive (interruptible) operating systems.
Note	Multitasking systems with a non-preemptive operating system do not require data integrity or determinism protection. In this case, always clear the parameters **Ensure data integrity during data transfer** and **Ensure deterministic data transfer**. Ensuring data integrity and determinism requires additional memory and execution time. To reduce this additional expense, evaluate all signals to determine the level of protection that they require.	
Prerequisites	cgsl_0205:Signal handling for multirate models	

ID: Title	**cgsl_0206: Data integrity and determinism in multitasking models**
See Also	• Rate Transition • “Data Transfer Problems”
Last Changed	R2011a

Configuration Parameter Considerations

cgsl_0301: Prioritization of code generation objectives for code efficiency

ID: Title	**cgsl_0301: Prioritization of code generation objectives for code efficiency**	
Description	Prioritize code generation objectives for code efficiency by using the Code Generation Advisor.	
	A	Assign priorities to code (`ROM`, `RAM`, and `Execution efficiency`) efficiency objectives.
	B	Select the relative order of `ROM`, `RAM`, and `Execution efficiency` based on application requirements.
	C	Configure the Code Generation Advisor to run before generating code by setting **Check model before generating code** on the **Code Generation** pane of the Configuration Parameters dialog box to `On (proceed with warnings)` or `On (stop for warnings)`.
Notes	A model's configuration parameters provide control over many aspects of generated code. The prioritization of objectives specifies how configuration parameters are set when conflicts between objectives occur. Prioritizing code efficiency objectives above safety objectives may remove initialization or run-time protection code (for example, saturation range checking for signals out of representable range). Review the resulting parameter configurations to verify that safety requirements are met. For more information about objective tradeoffs for each model parameter, see "Application Considerations" in the Embedded Coder™ documentation.	
Rationale	A, B, C	When you use the Code Generation Advisor, configuration parameters conform to the objectives that you want and they are consistently enforced.
See also	• "Set Objectives — Code Generation Advisor Dialog Box" in the Simulink Coder documentation • "Manage a Configuration Set" in the Simulink documentation • "hisl_0055: Prioritization of code generation objectives for high-integrity systems"	
Last Changed	R2010b	

cgsl_0302: Diagnostic settings for multirate and multitasking models

ID: Title	cgsl_0302: Diagnostic settings for multirate and multitasking models
Description	For multirate models using either **single tasking** or **multitasking**, set to either `warning` or `error` the following diagnostics: • **Diagnostics > Sample Time > Single task rate transition** • **Diagnostics > Sample Time > Enforce sample time specified by Signal Specification blocks** • **Diagnostics > Data Validity > Merge Block > Detect multiple driving blocks executing at the same time step** For **multitasking** models, set to either `warning` or `error` the following diagnostics: • **Diagnostics > Sample Time > Multitask task rate transition** • **Diagnostics > Sample Time > Multitask conditionally executed subsystem** • **Diagnostics > Sample Time >Tasks with equal priority** If the model contains Data Store Memory blocks, set to either `Enable all as warnings` or `Enable all as errors` the following diagnostics: • **Diagnostics > Data Validity > Data Store Memory Block > Detect read before write** • **Diagnostics > Data Validity > Data Store Memory Block > Detect write after read** • **Diagnostics > Data Validity > Data Store Memory Block > Detect write after write** • **Diagnostics > Data Validity > Data Store Memory Block > Multitask data store**
Rationale	Setting the diagnostics improves run-time detection of rate and tasking errors.

ID: Title	cgsl_0302: Diagnostic settings for multirate and multitasking models
See Also	• "Diagnostics Pane: Solver" • "hisl_0013: Usage of data store blocks"
Last Changed	2011a