Simulink®

Modeling Guidelines for Code Generation

MATLAB® & SIMULINK®

R2016a

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)
March 2012	Online only	Revised for Version 1.3 (Release 2012a)
September 2012	Online only	Revised for Version 1.4 (Release 2012b)
March 2013	Online only	Revised for Version 1.5 (Release 2013a)
September 2013	Online only	Revised for Version 1.6 (Release 2013b)
March 2014	Online only	Revised for Version 1.7 (Release 2014a)
October 2014	Online only	Revised for Version 1.8 (Release 2014b)
March 2015	Online only	Revised for Version 1.9 (Release 2015a)
September 2015	Online only	Revised for Version 1.10 (Release 2015b)
March 2016	Online only	Revised for Version 1.11 (Release 2016a)

Contents

Introduction

Motivation

MathWorks® intends the guidelines for engineers developing models and generating code for embedded systems using Model-Based Design with MathWorks products. The guidelines provide recommendations for model settings, block usage, and block parameters that impact simulation behavior or code generated by the Embedded Coder® product.

The guidelines do not address model style or development processes. For more information about creating models in a way that improves consistency, clarity, and readability, see the "MAAB Control Algorithm Modeling". Development process guidance and additional information for specific standards is available with the IEC Certification Kit (for ISO 26262 and IEC 61508) and DO Qualification Kit (for DO-178) products.

Disclaimer While adhering to the recommendations in the guidelines 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 the guidelines are not followed, it does not mean that the system being developed will be unsafe.

Guideline Template

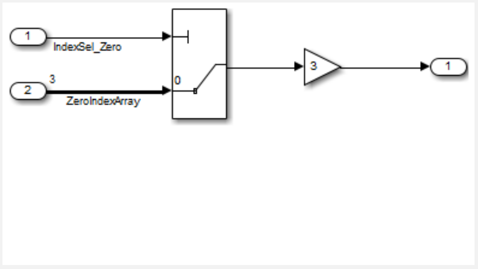
Guideline descriptions are documented, using the following template. Companies that want to create additional guidelines are encouraged to use the same template.

ID: Title	*XX_nnnn*: Title of the guideline (unique, short)
Description	Description of the guideline
Prerequisites	Links to guidelines that are prerequisites to this guideline (ID: Title)
Notes	Notes for using the guideline
Rationale	Rational for providing the guideline
Model Advisor Check	Title of and link to the corresponding Model Advisor check, if a check exists
References	References to standards that apply to guideline
See Also	Links to additional information
Last Changed	Version number of last change
Examples	Guideline examples

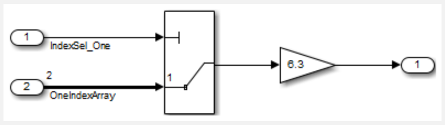
Block Considerations

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 contiguous** 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	 1
	IndexSel_Zero
	2
	3
	ZeroIndexArray
	0
	3
	1
	Recommended
	`void ZeroIndex(void)`
	`{`
	`  Y.Out5 = 3.0 * ZeroIndexArray[IndexSel_Zero];`
	`}`	

ID: Title	cgsl_0101: Zero-based indexing

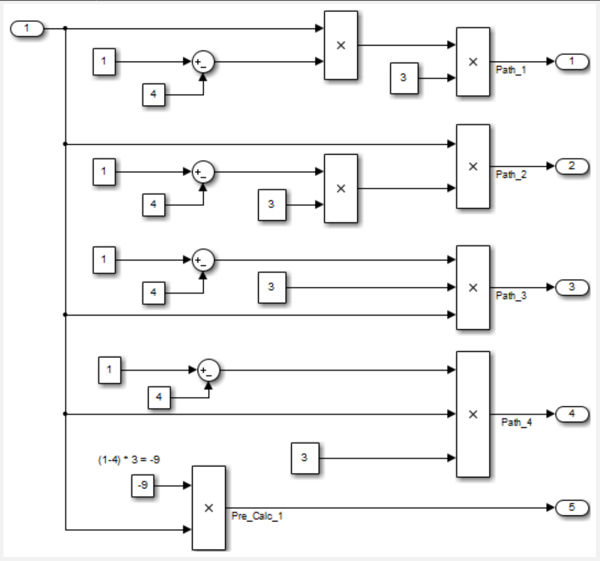
	 1
	IndexSel_One
	2
	2
	OneIndexArray
	1
	6.3
	1 **Not Recommended**
	`void OneIndex(void)`
	`{`
	`  Y.Out1 = OneIndexArray[IndexSel_One - 1] * 6.3;`
	`}`

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** For check details, see "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	Set the following model optimization parameters: • Set **Optimization** > **Signals and Parameters** > **Default parameter behavior** to `Inlined` • Enable **Optimization** > **Signals and Parameters** > **Code generation** > **Signals** > **Inline invariant signals**
Notes	Precalculating variables can reduce local and global memory usage and improve execution speed. If you set **Default parameter behavior** to `Inlined` and enable **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	R2012b	
Examples	In the following model, the 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
	 Path_1 Path_2 Path_3 Path_4 (1-4) * 3 = -9 Pre_Calc_1

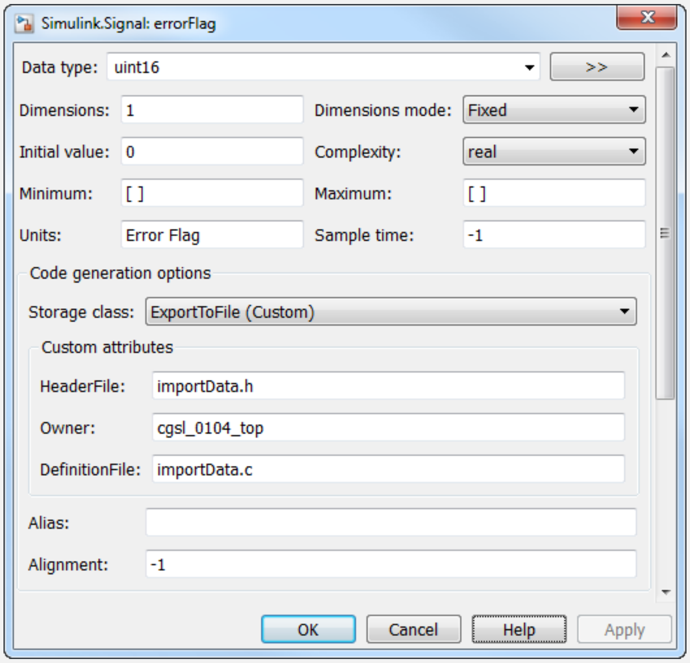
```
Path_1 = InputSignal * -3.0 * 3.0;

/* 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;

/* Product: '<Root>/Product6' incorporates:
 *  Constant: '<Root>/Constant3'
 *  Inport: '<Root>/In1'
```

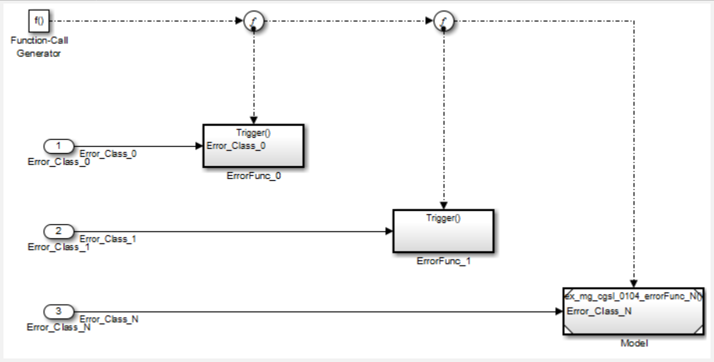
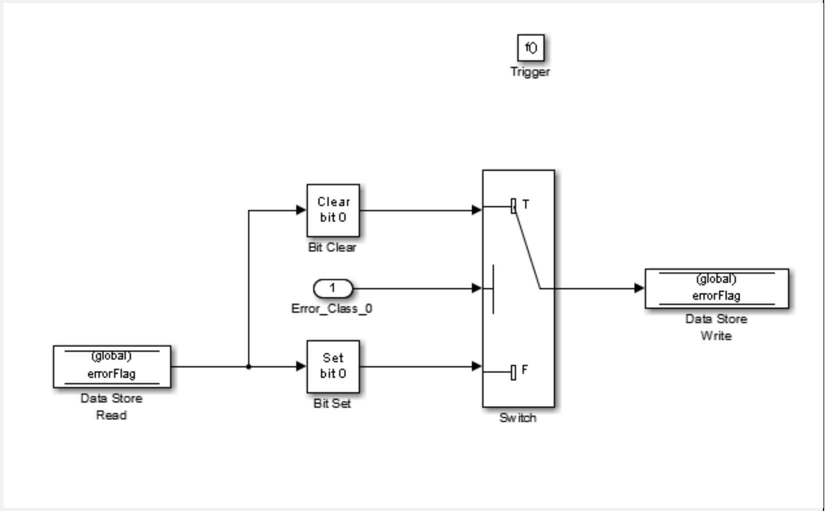
ID: Title	cgsl_0103: Precalculated signals and parameters

	` */`
	`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 “Parameter Storage in the Generated Code” 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 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 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 models and a single global instance is created in the generated code.
See Also	• "hisl_0013: Usage of data store blocks" • "hisl_0015: Usage of Merge blocks" • "cgsl_0302: Diagnostic settings for multirate and multitasking models" on page 4-3 • "cgsl_0105: Modeling local shared memory using data stores" on page 2-12	
Last Changed	R2011b	

ID: Title	cgsl_0104: Modeling global shared memory using data stores
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 signal data object.  Simulink.Signal: errorFlag Data type: uint16 Dimensions: 1 Dimensions mode: Fixed Initial value: 0 Complexity: real Minimum: [ ] Maximum: [ ] Units: Error Flag Sample time: -1 Code generation options Storage class: ExportToFile (Custom) Custom attributes HeaderFile: importData.h Owner: cgsl_0104_top DefinitionFile: importData.c Alias: Alignment: -1

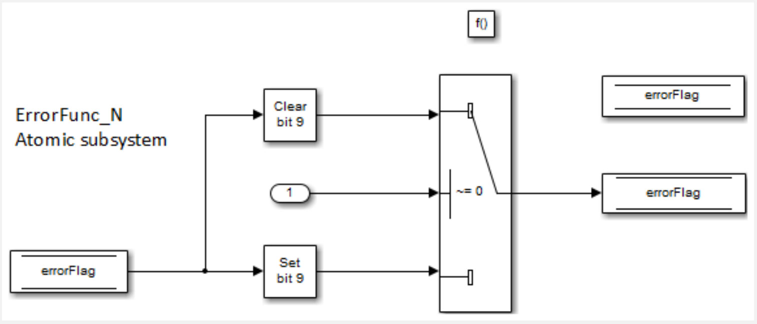
ID: Title	cgsl_0104: Modeling global shared memory using data stores
	 f() Function-Call Generator Trigger() Error_Class_0 ErrorFunc_0 1 Error_Class_0 Error_Class_0 Trigger() ErrorFunc_1 2 Error_Class_1 Error_Class_1 ex_mg_cgsl_0104_errorFunc_N() Error_Class_N Model 3 Error_Class_N Error_Class_N **Recommended** In this example, there are no `Data Store Memory` blocks. The resulting code uses the same global variable for the full model.  f() Trigger Clear bit 0 Bit Clear 1 Error_Class_0 Set bit 0 Bit Set T F Switch (global) errorFlag Data Store Read (global) errorFlag Data Store Write

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

```
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.

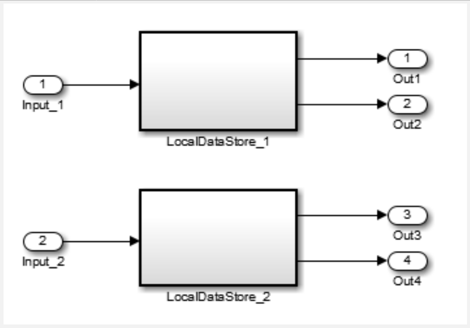


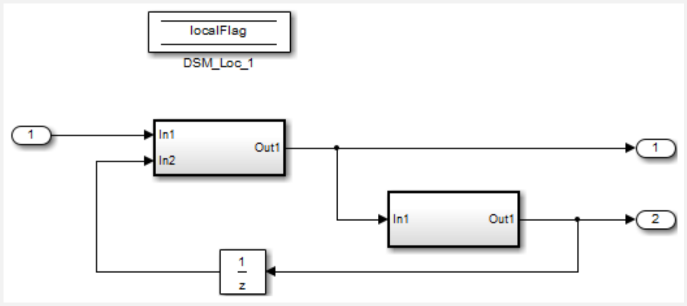
```
rtMdlrefDWork_mr_cgsl_0104_erro mr_cgsl_0104_errorF_MdlrefDWork;
void mr_cgsl_0104_errorFunc_N_UseDSM(const boolean_T *rtu_Error_Class_N)
{
  rtDW_mr_cgsl_0104_errorFunc_N_U *localDW =
    &(mr_cgsl_0104_errorF_MdlrefDWork.rtdw);
  if (*rtu_Error_Class_N) {
    localDW->errorFlag = (uint16_T)(~((uint16_T)(((uint16_T)(~localDW->errorFlag))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

cgsl_0105: Modeling local shared memory using data stores

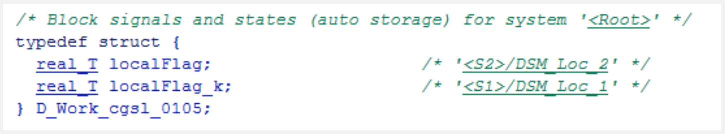
ID: Title	**cgsl_0105: Modeling local shared memory using data stores**	
Description	When using data store blocks as local shared memory:	
	A	Explicitly create the data store using a `Data Store Memory` block.
	B	Deselect the block parameter option **Data store name must resolve to Simulink signal object**.
	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" on page 2-8 • "cgsl_0302: Diagnostic settings for multirate and multitasking models" on page 4-3 • "hisl_0013: Usage of data store blocks"	
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.	

ID: Title	cgsl_0105: Modeling local shared memory using data stores
	The diagrams, text and code below





The instance of `localFlag` is in scope within the subsystem `LocalDataStore_1` and its subsystems.



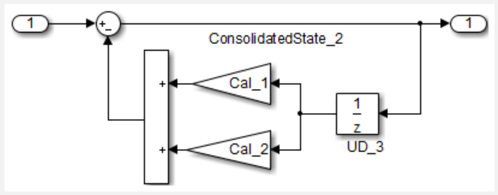
```
/* Block signals and states (auto storage) for system '<Root>' */
typedef struct {
  real_T localFlag;                    /* '<S2>/DSM_Loc_2' */
  real_T localFlag_k;                  /* '<S1>/DSM_Loc_1' */
} D_Work_cgsl_0105;
```

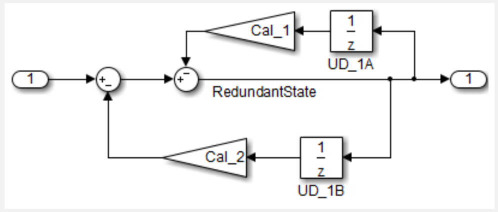
In the generated code, the data stores are part of the global DWork structure for the model. Embedded Coder automatically assigns them unique names during the code generation process.

Modeling Pattern Considerations

cgsl_0201: Redundant Unit Delay and Memory blocks

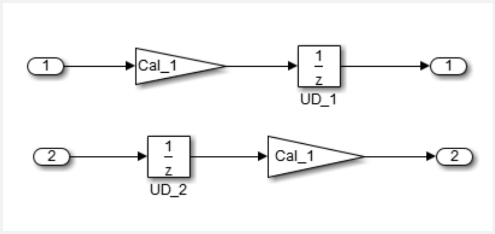
ID: Title	cgsl_0201: Redundant Unit Delay and Memory blocks	
Description	When preparing a model for code generation,	
	A	Remove redundant Unit Delay and Memory blocks.
Rationale	A	Redundant Unit Delay and Memory blocks use additional global memory. Removing the redundancies from a model reduces memory usage without impacting model behavior.
Last Changed	R2013a	

Example	 1
	1
	ConsolidatedState_2
	Cal_1
	Cal_2
	1/z
	UD_3 **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;`
	`}`	

	 Cal_1
	1/z
	UD_1A
	1
	1
	RedundantState
	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;`
	`}`	

ID: Title	cgsl_0201: Redundant Unit Delay and Memory blocks

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 a new position in the equation without changing the result.



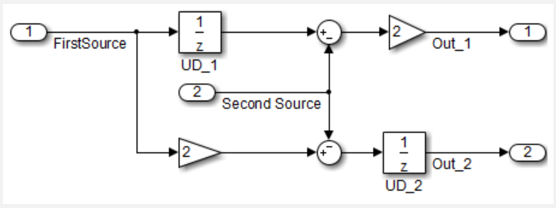
For the top path in the preceding example, the equations for the blocks are:

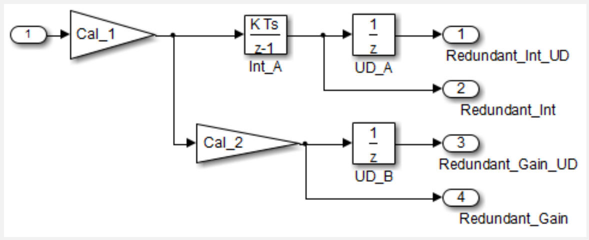
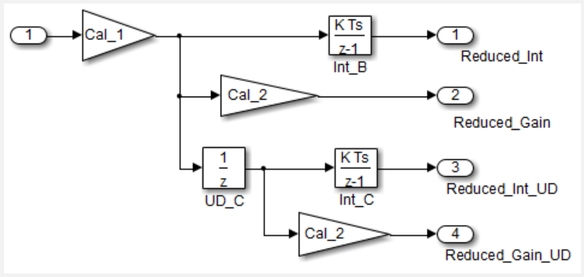
```
1  Out_1(t) = UD_1(t)
2  UD_1(t) = In_1(t-1) * Cal_1
3  Out_1(t) = In_1(t-1) * Cal_1
```

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 impacts the result. As the following example shows, the location of the Unit Delay block impacts the results due the skewing of the time sample between the top and bottom paths.



ID: Title	cgsl_0201: Redundant Unit Delay and Memory blocks
	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.   From a black box perspective, the two models are equivalent. However, from a memory and computation perspective, differences exist between the two models.

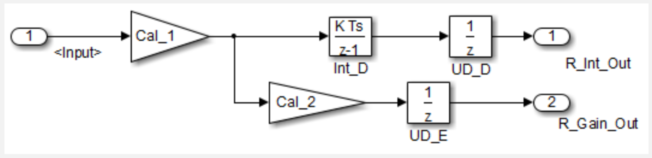
```
{
  real_T rtb_Gain4;
  rtb_Gain4 = Cal_1 * Redundant;
  Y.Redundant_Gain = Cal_2 * rtb_Gain4;
  Y.Redundant_Int = DWork.Int_A;
  Y.Redundant_Int_UD = DWork.UD_A;
  Y.Redundant_Gain_UD = DWork.UD_B;
  DWork.Int_A = 0.01 * rtb_Gain4 + DWork.Int_A;
  DWork.UD_A = Y.Redundant_Int;
  DWork.UD_B = Y.Redundant_Gain;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

{
  real_T rtb_Gain1;
  real_T rtb_UD_C;
```

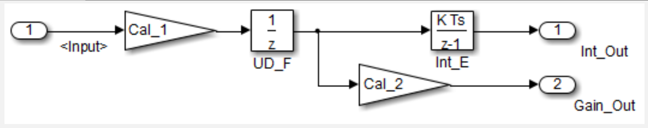
ID: Title	cgsl_0201: Redundant Unit Delay and Memory blocks

```
  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;
  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 discrete 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.



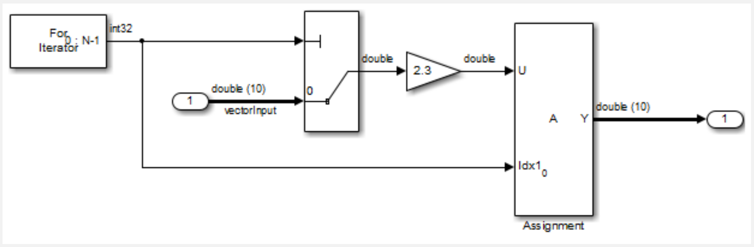
ID: Title	cgsl_0201: Redundant Unit Delay and Memory blocks
	 1 <Input> Cal_1 1/z UD_F K Ts/z-1 Int_E Cal_2 1 Int_Out 2 Gain_Out

```
{
  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;
}
```

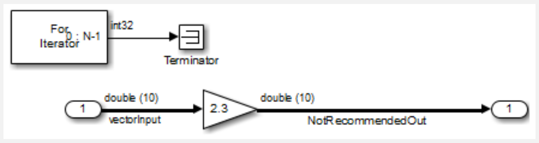
The code for the refactored model is more efficient because the branches from the root signal do not have a redundant unit delay.

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
	int32
	double (10)
	1
	vectorInput
	0
	double
	2.3
	double
	U
	A
	Y
	Idx1
	double (10)
	1
	Assignment
	Recommended
	`for (s1_iter = 0; s1_iter < 10; s1_iter++) {`
	`  RecommendedOut[s1_iter] = 2.3 * vectorInput[s1_iter];`
	`}`
	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.	

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

	 For Iterator
	0 : N-1
	int32
	Terminator
	1
	double (10)
	vectorInput
	2.3
	double (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 or Model blocks

ID: Title	cgsl_0204: Vector and bus signals crossing into atomic subsystems or Model blocks
Description	When working with vector or bus signals and some of the signal elements are in an atomic subsystem or a referenced model, use the following information to determine how to select signal elements to minimize memory usage.

A **Bus or vector entering an atomic subsystem:**

Function packaging: `Non-reusable function`

Function interface: `void_void`

	Signals selected outside subsystem results in...	**Signal selected inside subsystem results in...**
Virtual Bus	No data copies.	No data copies.
Nonvirtual Bus	No data copies.	No data copies.
Vector	A copy of the selected signals in global block I/O structure that is used in the function.	No data copies.

Function packaging: `Non-reusable function`

Function interface: `Allow arguments`

	Signals selected outside subsystem results in	**Signal selected inside subsystem results in**
Virtual Bus	No data copies. Only the selected signals are passed to the function.	No data copies. Only the selected signals are passed to the function.

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

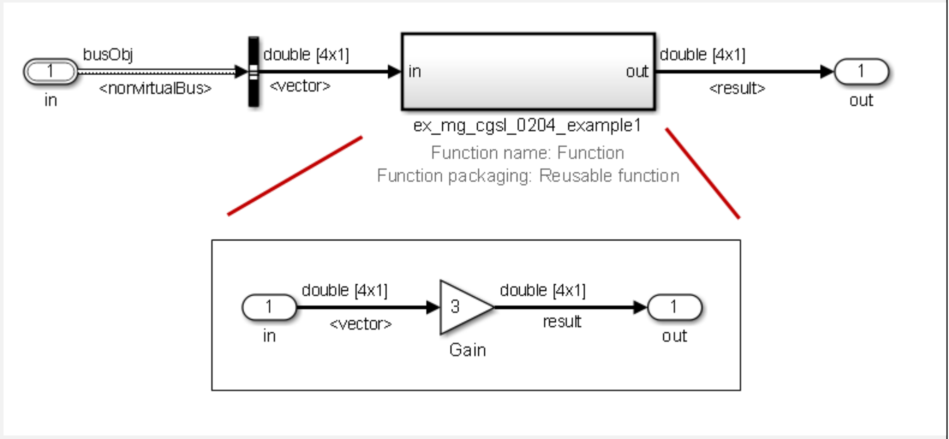
Nonvirtual Bus	No data copies. Only the selected signals are passed to the function.	No data copies. The whole bus is passed to the function.
Vector	A copy of the selected signals in a local variable that is passed to the function.	No data copies. The whole vector is passed to the function.

Function packaging: `Reusable function`

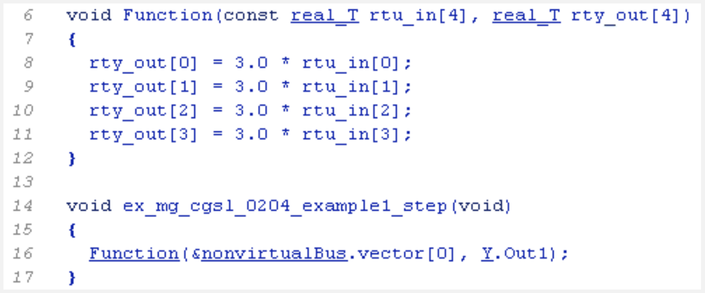
	Signals selected outside subsystem results in	**Signal selected inside the subsystem results in**
Virtual Bus	No data copies. Only the selected signals are passed to the function.	No data copies. Only the selected signals are passed to the function.
Nonvirtual Bus	No data copies. Only the selected signals are passed to the function. See Example 1.	No data copies. The whole bus is passed to the function.
Vector	A copy of the selected signals in a local variable that is passed to the function.	No data copies. The whole vector is passed to the function.

ID: Title	cgsl_0204: Vector and bus signals crossing into atomic subsystems or Model blocks	
	B	**Bus or vector entering a Model block:**

	Signals selected outside Model block results in...	**Signal selected inside Model block results in...**
Virtual Bus	No data copies. Only selected signals are passed to the function.	If `Inport` block parameter **Output as virtual bus** is selected, then there are no data copies. Only the selected signals are passed to the function. If `Inport` block parameter **Output as virtual bus** is cleared, then a copy of the whole bus is passed to the function.
Nonvirtual Bus	No data copies. Only the selected signals are passed to the function.	If `Inport` block parameter **Output as virtual bus** is selected, then there are no data copies. Only the selected signals are passed to the function. If `Inport` block parameter **Output as virtual bus** is cleared, then a copy of the whole bus is passed to

ID: Title	**cgsl_0204: Vector and bus signals crossing into atomic subsystems or Model blocks**			
				the function. See Example 2.
		Vector	A copy of the selected signals in a local variable that is passed to the function.	No data copies. The whole vector is passed to the function.
Notes	• Depending on Embedded Coder settings (e.g. optimizations), predecessor blocks and signal storage classes, actual results may differ from the tables. • Virtual busses do not support global data. • If the subsystem is set to `Inline`, data copies do not occur.			
Rationale	A, B	Minimize RAM, ROM, and stack usage		
Last Changed	R2016a			
Examples	**Example 1**: Nonvirtual bus entering an atomic subsystem • **Function packaging:** `Reusable function` • Selection: Sub-signal selected outside the subsystem  busObj 1 in <nonvirtualBus> double [4x1] <vector> in out double [4x1] <result> 1 out ex_mg_cgsl_0204_example1 Function name: Function Function packaging: Reusable function 1 in double [4x1] <vector> 3 Gain double [4x1] result 1 out			

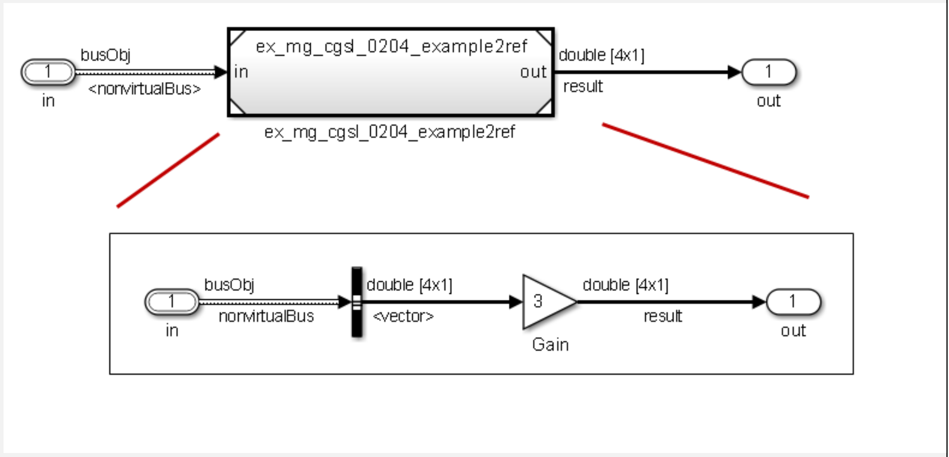
ID: Title	cgsl_0204: Vector and bus signals crossing into atomic subsystems or Model blocks
	Only the selected signals are passed to the function:



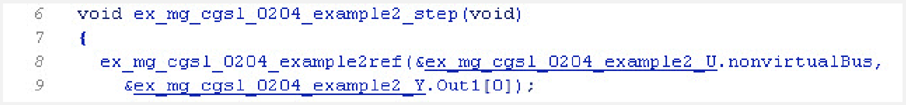
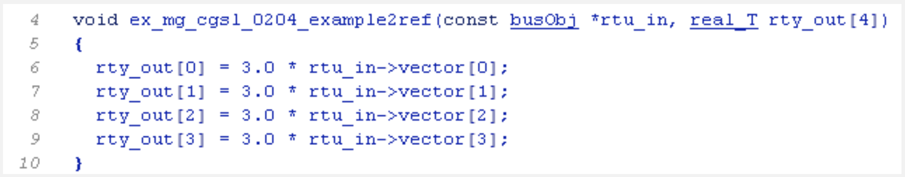
```
void Function(const real_T rtu_in[4], real_T rty_out[4])
{
  rty_out[0] = 3.0 * rtu_in[0];
  rty_out[1] = 3.0 * rtu_in[1];
  rty_out[2] = 3.0 * rtu_in[2];
  rty_out[3] = 3.0 * rtu_in[3];
}

void ex_mg_cgsl_0204_example1_step(void)
{
  Function(&nonvirtualBus.vector[0], Y.Out1);
}
```

Example 2: Nonvirtual bus entering a model block

- **Total number of instances allowed per top model:** `Multiple`
- Selection: Sub-signal selected inside the referenced model

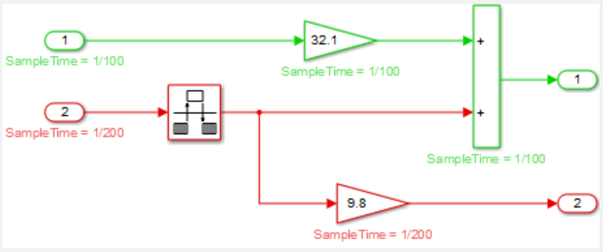
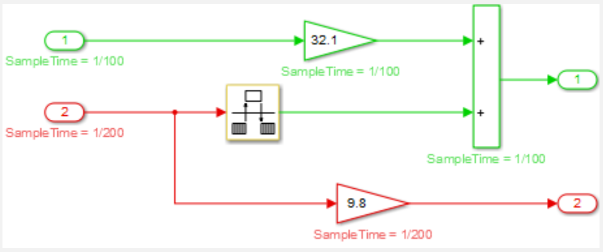


ID: Title	cgsl_0204: Vector and bus signals crossing into atomic subsystems or Model blocks

	There are no data copies in the code for the main model. The whole bus is passed to the model reference function.
	
	`void ex_mg_cgsl_0204_example2_step(void)`
	`{`
	`  ex_mg_cgsl_0204_example2ref(&ex_mg_cgsl_0204_example2_U.nonvirtualBus,`
	`    &ex_mg_cgsl_0204_example2_Y.Out1[0]);`
	
	Code for the model reference function:
	
	`void ex_mg_cgsl_0204_example2ref(const busObj *rtu_in, real_T rty_out[4])`
	`{`
	`  rty_out[0] = 3.0 * rtu_in->vector[0];`
	`  rty_out[1] = 3.0 * rtu_in->vector[1];`
	`  rty_out[2] = 3.0 * rtu_in->vector[2];`
	`  rty_out[3] = 3.0 * rtu_in->vector[3];`
	`}`

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 supports the 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 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.	
Last Changed	R2011a	
Examples	**Not Recommended:** 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	

ID: Title	cgsl_0205: Signal handling for multirate models
	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.  1 SampleTime = 1/100 2 SampleTime = 1/200 32.1 SampleTime = 1/100 + + SampleTime = 1/100 1 9.8 SampleTime = 1/200 2 **Recommended:** In this example, the rate transition is inserted at the destination of the signal.  1 SampleTime = 1/100 2 SampleTime = 1/200 32.1 SampleTime = 1/100 + + SampleTime = 1/100 1 9.8 SampleTime = 1/200 2

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`.
Prerequisites	“cgsl_0205: Signal handling for multirate models” on page 3-15	
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, 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 signals to determine the level of protection that they require.	
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 in 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.	
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	• "Application Objectives Using Code Generation Advisor" 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	R2015b	

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 > All Parameters > 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.
See Also	• "Diagnostics Pane: Solver" • "hisl_0013: Usage of data store blocks" • "hisl_0044: Configuration Parameters > Diagnostics > Sample Time"

ID: Title	cgsl_0302: Diagnostic settings for multirate and multitasking models
Last Changed	2016a