Simulink®

Modeling Guidelines for Code Generation

MATLAB® & SIMULINK®

R2022b

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Modeling Guidelines for Code Generation

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)
September 2016	Online only	Revised for Version 1.12 (Release 2016b)
March 2017	Online only	Revised for Version 1.13 (Release 2017a)
September 2017	Online only	Revised for Version 1.14 (Release 2017b)
March 2018	Online only	Revised for Version 1.15 (Release 2018a)
September 2018	Online only	Revised for Version 1.16 (Release 2018b)
March 2019	Online only	Revised for Version 1.17 (Release 2019a)
September 2019	Online only	Revised for Version 1.18 (Release 2019b)
March 2020	Online only	Revised for Version 1.19 (Release 2020a)
September 2020	Online only	Revised for Version 1.20 (Release 2020b)
March 2021	Online only	Revised for Version 1.21 (Release 2021a)
September 2021	Online only	Revised for Version 1.22 (Release 2021b)
March 2022	Online only	Revised for Version 1.23 (Release 2022a)
September 2022	Online only	Revised for Version 1.24 (Release 2022b)

Contents

4 Configuration Parameter Considerations

5 Component Deployment Using Service Interface Configuration

Introduction

Motivation

Code generation modeling guidelines provide recommendations that you can use when developing models and generating code that is intended for use in embedded systems. The guidelines, which take into consideration the potential impact to simulation behavior, code generation, and component model deployment, include information about configuration settings, block usage and parameters, and modeling patterns.

The guidelines do not address model style or compliance with industry standards. For additional information, see:

- “MAB Modeling Guidelines” — Modeling guidelines that address model consistency, clarity, and readability.
- “High-Integrity System Modeling” — Modeling guidelines that address compliance with industry standards.

For information about qualifying software development and verification tools that are used to develop embedded system for projects that must comply with an industry standard, see:

- IEC Certification Kit — Guidance on certifying your embedded systems for use in projects that must comply with ISO 26262, IEC 61508, EN 50128, EN 50657, ISO 25119, and related functional-safety standards such as IEC 62304.
- “DO Qualification Kit (for DO-178)” — Guidance on qualifying your software verification tools for use in projects involving the DO-178C and DO-254 standards.

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

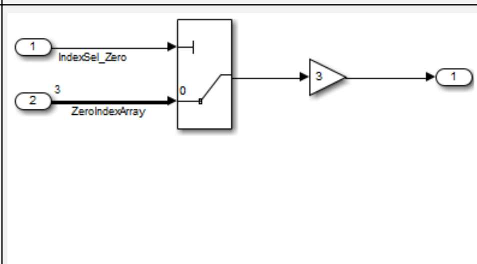
Guideline descriptions are documented, using the following template. Companies that want to create additional guidelines are encouraged to use the same template.

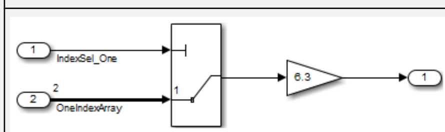
ID: Title	*XX_nnnn*: Title of the guideline (unique, short)
Description	Description of the guideline
Prerequisites	Links to guidelines that are prerequisites to this guideline (ID: Title)
Notes	Notes for using the guideline
Rationale	Rationale for providing the guideline
Model Advisor Check	Title of 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	For the Index Vector block parameter **Data port order**, select `Zero-based contiguous`.
	B	Set block parameter **Index mode** to `Zero-based` for the following blocks: • Assignment • Selector • For Iterator • Find Nonzero Elements
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];`
	`}`	

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

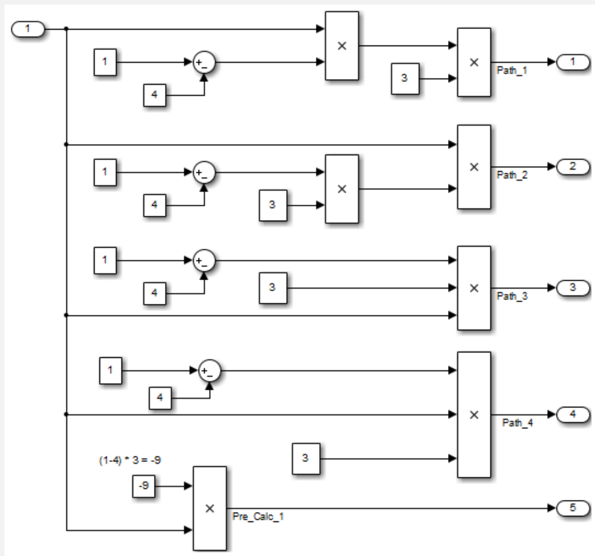
cgsl_0102: Evenly spaced breakpoints in lookup tables

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	**By Product > Embedded Coder > Identify questionable fixed-point operations** For check details, see “Identify questionable fixed-point operations” (Embedded Coder).	
See Also	“Formulation of Evenly Spaced Breakpoints”	
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 these configuration parameters: • Set **Default parameter behavior** to `Inlined` • Select **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 run time. 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	

ID: Title	cgsl_0103: Precalculated signals and parameters
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.




```
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'
 */
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

ID: Title	cgsl_0103: Precalculated signals and parameters
	information, see “Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder).

cgsl_0104: Modeling global shared memory using data stores

<table>
<tr><th>ID: Title</th><th colspan="2">cgsl_0104: Modeling global shared memory using data stores</th></tr>
<tr><td rowspan="4">Description</td><td colspan="2">When using data store blocks to model shared memory across multiple models:</td></tr>
<tr><td>A</td><td>Set configuration parameters Duplicate data store names to <code>error</code> for models in the hierarchy.</td></tr>
<tr><td>B</td><td>Define the data store using a Simulink Signal or MPT Signal object.</td></tr>
<tr><td>C</td><td>Do not use Data Store Memory blocks in the model.</td></tr>
<tr><td>Notes</td><td colspan="2">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.

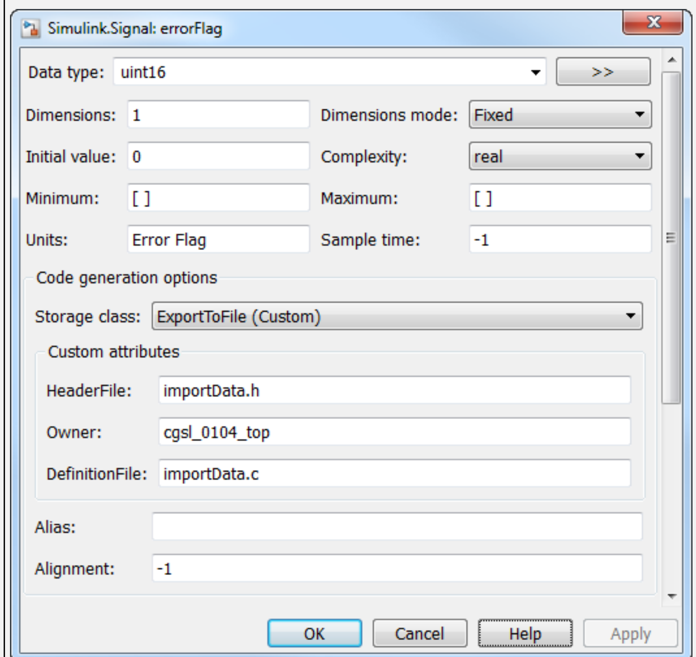
Use Duplicate data store names to help detect unintended identifier reuse. For models intentionally using local data stores, set the diagnostic to <code>warning</code>. Verify that only intentional data stores are included.

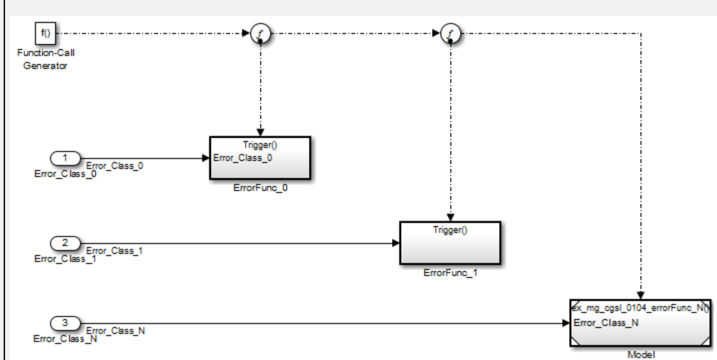
Merge blocks, used in conjunction with subsystems operating in a mutually exclusive manor, provide a second method of modeling global data across multiple models.</td></tr>
<tr><td>Rationale</td><td>A, B, C</td><td>Promotes a modeling pattern where a single consistent data store is used across models and a single global instance is created in the generated code.</td></tr>
<tr><td>See Also</td><td colspan="2">• “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 2-10</td></tr>
<tr><td>Last Changed</td><td colspan="2">R2011b</td></tr>
</table>

<table>
<tr><th>ID: Title</th><th>cgsl_0104: Modeling global shared memory using data stores</th></tr>
<tr><td>Examples</td><td>

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.



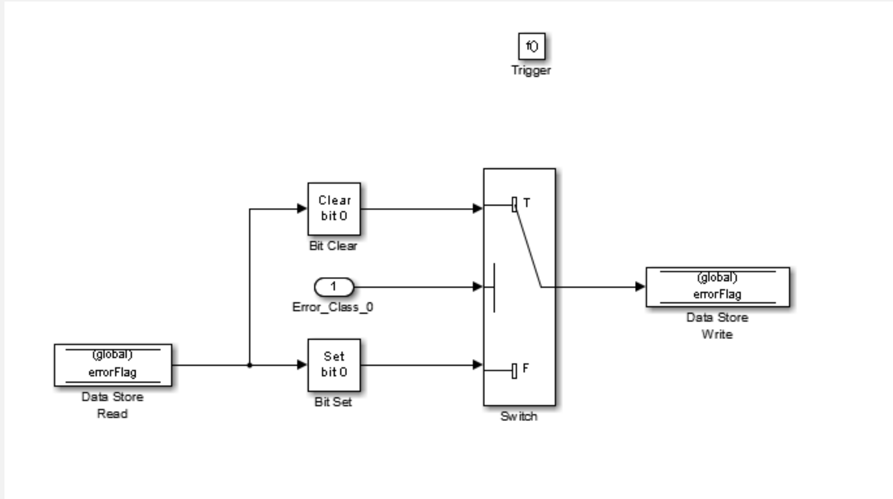


Recommended

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

</td></tr>
</table>

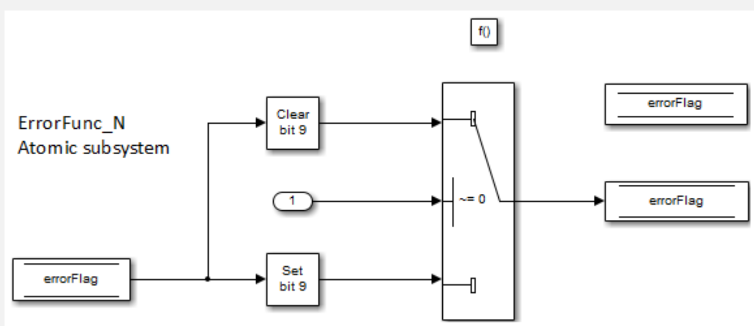
ID: Title	cgsl_0104: Modeling global shared memory using data stores




```
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.



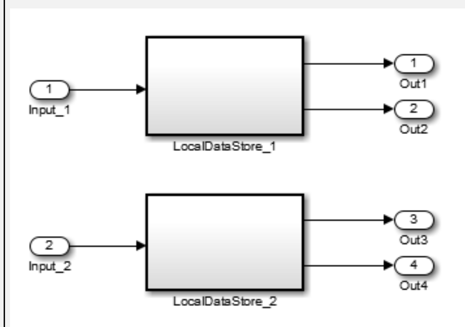

```
rtMdlrefDWork_mr_cgsl_0104_erro mr_cgsl_0104_errorF_MdlrefDWork;
void mr_cgsl_0104_errorFunc_N_UseDSM(const boolean_T *rtu_Error_Class_N)
{
  rtDW_mr_cgsl_0104_errorFunc_N_U *localDW =
    &(mr_cgsl_0104_errorF_MdlrefDWork.rtdw);
  if (*rtu_Error_Class_N) {
    localDW->errorFlag = (uint16_T)(~((uint16_T)(((uint16_T)(~localDW->errorFlag))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

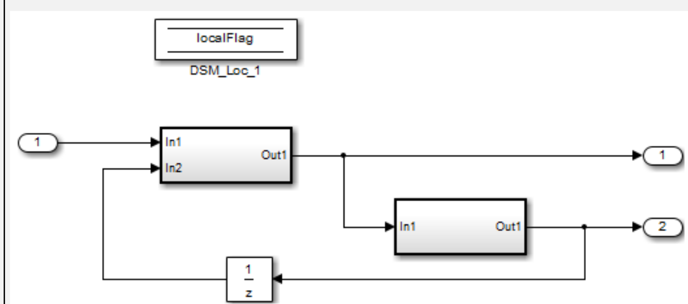
cgsl_0105: Modeling local shared memory using data stores

<table>
<tr><th>ID: Title</th><th colspan="2">cgsl_0105: Modeling local shared memory using data stores</th></tr>
<tr><td rowspan="4">Description</td><td colspan="2">When using data store blocks as local shared memory:</td></tr>
<tr><td>A</td><td>Explicitly create the data store using a Data Store Memory block.</td></tr>
<tr><td>B</td><td>Clear block parameter Data store name must resolve to Simulink signal object.</td></tr>
<tr><td>C</td><td>Consider following a naming convention for local Data Store Memory blocks.</td></tr>
<tr><td>Notes</td><td colspan="2">Use configuration parameter Duplicate data store names to help detect unintended identifier reuse. For models intentionally using local data stores, set the diagnostic to <code>warning</code>. 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.</td></tr>
<tr><td rowspan="2">Rationale</td><td>A, B</td><td>Data store block is treated as a local instance of the data store</td></tr>
<tr><td>C</td><td>Provides graphical feedback that the data store is local</td></tr>
<tr><td>See Also</td><td colspan="2">• “cgsl_0104: Modeling global shared memory using data stores” on page 2-7
• “cgsl_0302: Diagnostic settings for multirate and multitasking models” on page 4-3
• “hisl_0013: Usage of data store blocks”</td></tr>
<tr><td>Last Changed</td><td colspan="2">R2011b</td></tr>
</table>

<table>
<tr><th>ID: Title</th><th>cgsl_0105: Modeling local shared memory using data stores</th></tr>
<tr><td>Examples</td><td>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.





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

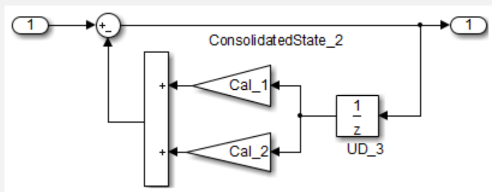
/* Block signals and states (auto storage) for system '<Root>' */
typedef struct {
  real_T localFlag; /* '<S2>/DSM_Loc_2' */
  real_T localFlag_k; /* '<S1>/DSM_Loc_1' */
} D_Work_cgsl_0105;

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.</td></tr>
</table>

Modeling Pattern Considerations

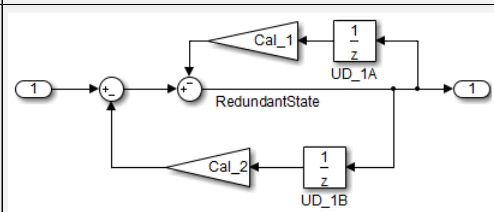
cgsl_0201: Redundant Unit Delay and Memory blocks

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		



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;
}
```

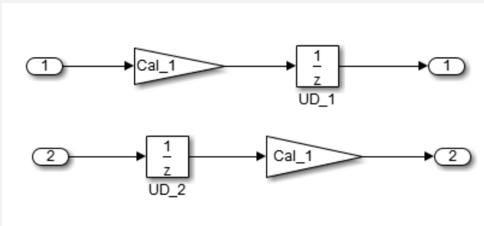


Not Recommended: Redundant Unit Delays

```
void Redundant(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.



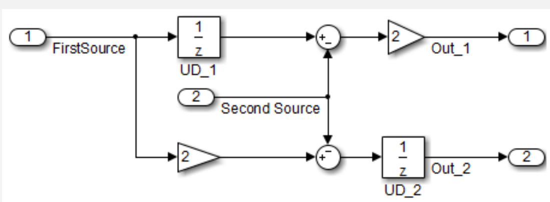
For the top path in the preceding example, the equations for the blocks are:

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

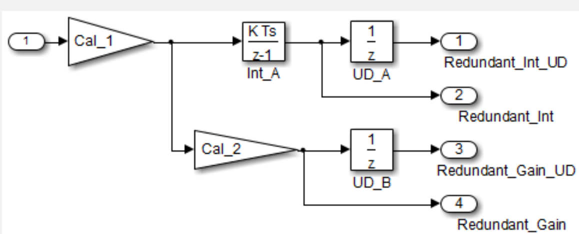
For the bottom path, the equations are:

1. `Out_2(t) = UD_2(t) * Cal_1`
2. `UD_2(t) = In_2(t-1)`

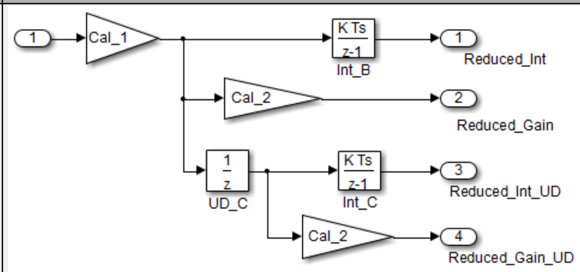
In contrast, if you add a secondary signal to the equations, the location of the Unit Delay block impacts the result. As the following example shows, the location of the Unit Delay block impacts the results due to the skewing of the time sample between the top and bottom paths.



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: Redundant Unit Delay and Memory blocks

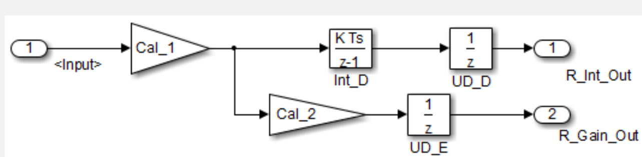


From a black box perspective, the two models are equivalent. However, from a memory and computation perspective, differences exist between the two models.

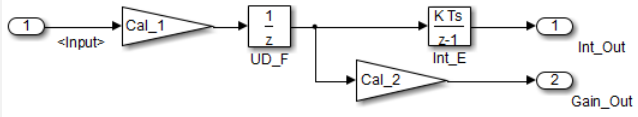
```
{
  real_T rtb_Gain4;
  rtb_Gain4 = Cal_1 * Redundant;
  Y.Redundant_Gain = Cal_2 * rtb_Gain4;
  Y.Redundant_Int = DWork.Int_A;
  Y.Redundant_Int_UD = DWork.UD_A;
  Y.Redundant_Gain_UD = DWork.UD_B;
  DWork.Int_A = 0.01 * rtb_Gain4 + DWork.Int_A;
  DWork.UD_A = Y.Redundant_Int;
  DWork.UD_B = Y.Redundant_Gain;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

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

In this case, the original model is more efficient. In the first code example, there are three global variables, 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 Discrete Time Integrator blocks (DWork.Int_B and DWork.Int_C). The Discrete 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.

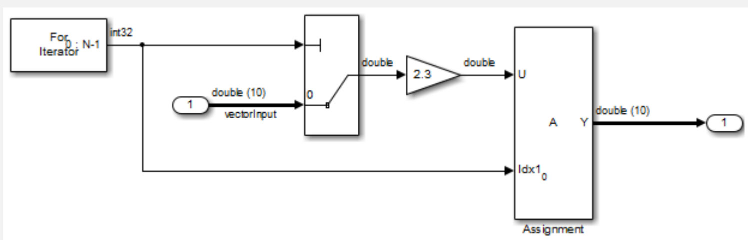


ID: Title	cgsl_0201: Redundant Unit Delay and Memory blocks

	
	`{`
	`  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

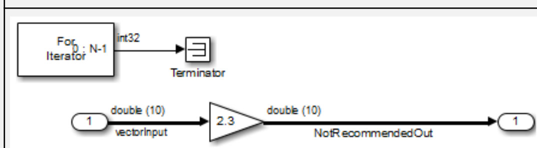
ID: Title	cgsl_0202: Usage of For, While, and For Each subsystems with vector signals	
Description	When developing a model for code generation,	
	A	Use For, While, and For Each subsystems for calculations that require iterative behavior or operate on a subset (frame) of data.
	B	Avoid using For, While,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



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.



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

<table>
<tr><th>ID: Title</th><th colspan="2">cgsl_0204: Vector and bus signals crossing into atomic subsystems or Model blocks</th></tr>
<tr><td rowspan="3">Description</td><td colspan="2">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.</td></tr>
<tr><td rowspan="2">A</td><td>Bus or vector entering an atomic subsystem:
<table>
<tr><td colspan="3">Function packaging: <code>Non-reusable function</code>

Function interface: <code>void_void</code></td></tr>
<tr><td></td><td>Signals selected outside subsystem results in...</td><td>Signal selected inside subsystem results in...</td></tr>
<tr><td>Virtual Bus</td><td>No data copies.</td><td>No data copies.</td></tr>
<tr><td>Nonvirtual Bus</td><td>No data copies.</td><td>No data copies.</td></tr>
<tr><td>Vector</td><td>A copy of the selected signals in global block I/O structure that is used in the function.</td><td>No data copies.</td></tr>
</table></td></tr>
<tr><td><table>
<tr><td colspan="3">Function packaging: <code>Non-reusable function</code>

Function interface: <code>Allow arguments (Optimized)</code></td></tr>
<tr><td></td><td>Signals selected outside subsystem results in</td><td>Signal selected inside subsystem results in</td></tr>
<tr><td>Virtual Bus</td><td>No data copies. Only the selected signals are passed to the function.</td><td>No data copies. Only the selected signals are passed to the function.</td></tr>
<tr><td>Nonvirtual Bus</td><td>No data copies. Only the selected signals are passed to the function.</td><td>No data copies. The whole bus is passed to the function.</td></tr>
<tr><td>Vector</td><td>A copy of the selected signals in a local variable that is passed to the function.</td><td>No data copies. The whole vector is passed to the function.</td></tr>
</table></td></tr>
</table>

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

Function packaging: `Reusable function`

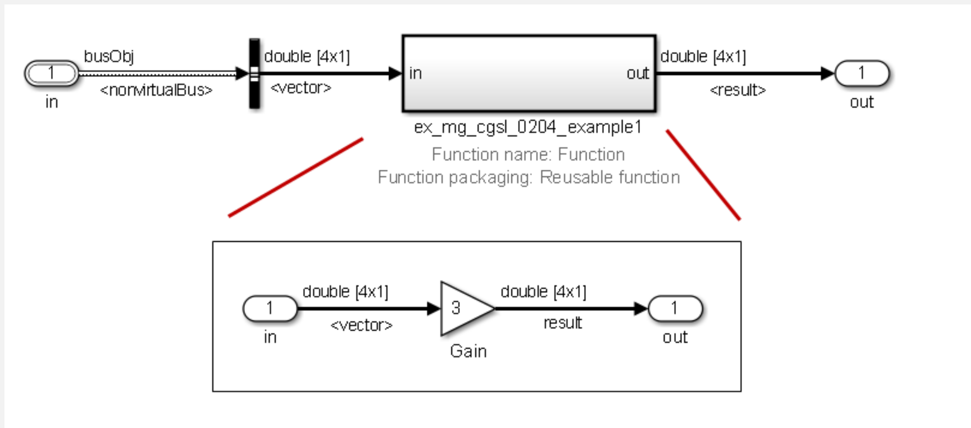
	Signals selected outside subsystem results in	Signal selected inside the subsystem results in
Virtual Bus	No data copies. Only the selected signals are passed to the function.	No data copies. Only the selected signals are passed to the function.
Nonvirtual Bus	No data copies. Only the selected signals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

<table>
<tr><th>ID: Title</th><th colspan="2">cgsl_0204: Vector and bus signals crossing into atomic subsystems or Model blocks</th></tr>
<tr><td></td><td>B</td><td>Bus or vector entering a Model block:
<table>
<tr><th></th><th>Signals selected outside Model block results in...</th><th>Signal selected inside Model block results in...</th></tr>
<tr><td>Virtual Bus</td><td>No data copies. Only selected signals are passed to the function.</td><td>If Inport block parameter Output as nonvirtual bus is selected, then there are no data copies. Only the selected signals are passed to the function.

If Inport block parameter Output as nonvirtual bus is cleared, then a copy of the whole bus is passed to the function.</td></tr>
<tr><td>Nonvirtual Bus</td><td>No data copies. Only the selected signals are passed to the function.</td><td>If Inport block parameter Output as nonvirtual bus is selected, then there are no data copies. Only the selected signals are passed to the function.

If Inport block parameter Output as nonvirtual bus is cleared, then a copy of the whole bus is passed to the function. See example 2.</td></tr>
<tr><td>Vector</td><td>A copy of the selected signals in a local variable that is passed to the function.</td><td>No data copies. The whole vector is passed to the function.</td></tr>
</table></td></tr>
<tr><td>Notes</td><td colspan="2">Depending on Embedded Coder settings (e.g. optimizations), predecessor blocks and signal storage classes, actual results might differ from the tables.
Virtual busses do not support global data.
If the subsystem is set to <code>Inline</code>, data copies do not occur.</td></tr>
<tr><td>Rationale</td><td>A, B</td><td>Minimize RAM, ROM, and stack usage</td></tr>
<tr><td>Last Changed</td><td colspan="2">R2016a</td></tr>
</table>

ID: Title	cgsl_0204: Vector and bus signals crossing into atomic subsystems or Model blocks
Examples	**Example 1**: Nonvirtual bus entering an atomic subsystem • **Function packaging:** `Reusable function` • Selection: Subsignal selected outside the subsystem



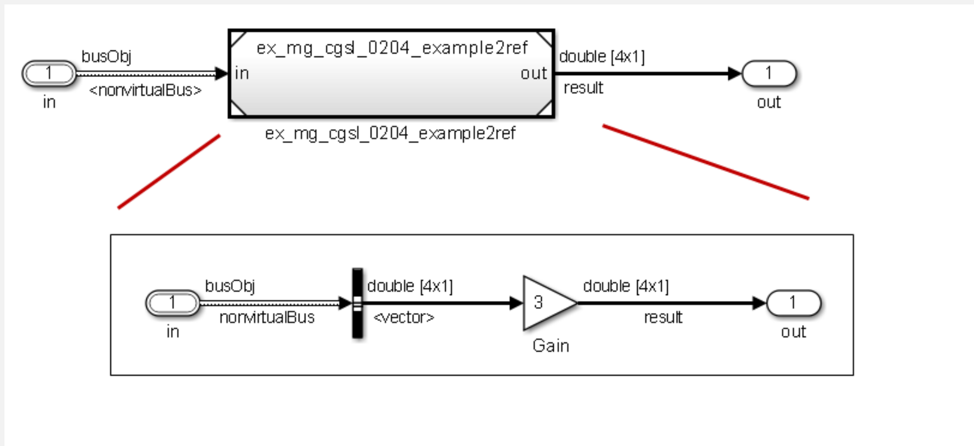
Only the selected signals are passed to the function:

```
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);
}
```

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

Example 2: Nonvirtual bus entering a model block

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



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

<table>
<tr><th>ID: Title</th><th colspan="2">cgsl_0205: Signal handling for multirate models</th></tr>
<tr><td rowspan="3">Description</td><td colspan="2">For multirate models, handle the change in operation rate in one of two ways:</td></tr>
<tr><td>A</td><td>At the destination block, Insert a Rate Transition.</td></tr>
<tr><td>B</td><td>Set configuration parameter Automatically handle rate transition for data transfer to <code>Always</code> or <code>Whenever possible</code>.</td></tr>
<tr><td>Rationale</td><td>A,B</td><td>Following this guideline supports the handling of data operating at different rates.</td></tr>
<tr><td>Note</td><td colspan="2">Setting Automatically handle rate transition for data transfer to <code>Whenever possible</code> requires you to insert a Rate Transition block in locations indicated by Simulink.

Setting Automatically handle rate transition for data transfer to <code>Always</code> 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.

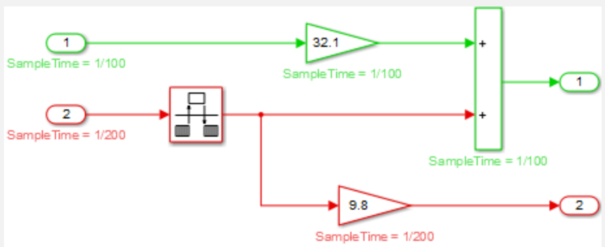
For these cases, manually insert a Rate Transition block or blocks.

MathWorks does not recommend using Unit Delay and Zero Order Hold blocks for handling rate transitions.</td></tr>
<tr><td>Last Changed</td><td colspan="2">R2011a</td></tr>
</table>

<table>
<tr><th>ID: Title</th><th>cgsl_0205: Signal handling for multirate models</th></tr>
<tr><td>Examples</td><td>

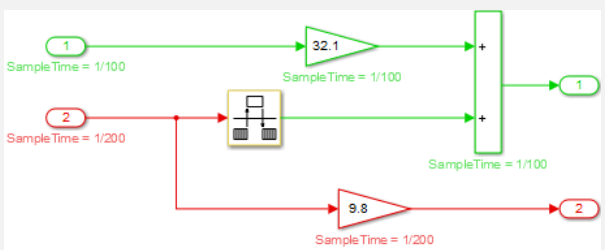
Not Recommended:

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 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.



Recommended:

In this example, the rate transition is inserted at the destination of the signal.



</td></tr>
</table>

cgsl_0206: Data integrity and determinism in multitasking models

<table>
<tr><th>ID: Title</th><th colspan="2">cgsl_0206: Data integrity and determinism in multitasking models</th></tr>
<tr><td rowspan="6">Description</td><td colspan="2">For multitasking models that are deployed with a preemptive (interruptible) operating system, protect the integrity of selected signals by doing one of the following:</td></tr>
<tr><td>A</td><td>Select the Rate Transition block parameter Ensure data integrity during data transfer .</td></tr>
<tr><td>B</td><td>For Inport blocks in Function Called subsystems, select the block parameter Latch input for feedback signals of function-call subsystem outputs.</td></tr>
<tr><td colspan="2">To protect selected signal determinism, do one of the following:</td></tr>
<tr><td>C</td><td>Select the Rate Transition block parameter Ensure deterministic data transfer (maximum delay).</td></tr>
<tr><td>D</td><td>• Select the configuration parameter Automatically handle rate transition for data transfer.
• Set configuration parameter Deterministic data transfer to <code>Whenever possible</code> or <code>Always</code>.</td></tr>
<tr><td>Prerequisites</td><td colspan="2">cgsl_0205:Signal handling for multirate models on page 3-12</td></tr>
<tr><td>Rationale</td><td>A,B, C,D</td><td>Following this guideline protects data against possible corruption of preemptive (interruptible) operating systems.</td></tr>
<tr><td>Note</td><td colspan="2">Multitasking systems with a non-preemptive operating system do not require data integrity or determinism protection. In this case, clear these parameters:
• Rate Transition block parameter Ensure data integrity during data transfer
• Configuration parameter Ensure deterministic data transfer (maximum delay)

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.</td></tr>
<tr><td>See Also</td><td colspan="2">• Rate Transition
• “Data Transfer Problems” (Simulink Coder)</td></tr>
<tr><td>Last Changed</td><td colspan="2">R2011a</td></tr>
</table>

Configuration Parameter Considerations

cgsl_0301: Prioritization of code generation objectives for code efficiency

<table>
<tr><th>ID: Title</th><th colspan="2">cgsl_0301: Prioritization of code generation objectives for code efficiency</th></tr>
<tr><td rowspan="4">Description</td><td colspan="2">Prioritize code generation objectives for code efficiency by using the Code Generation Advisor.</td></tr>
<tr><td>A</td><td>Assign priorities to code (`ROM`, `RAM`, and `Execution efficiency`) efficiency objectives.</td></tr>
<tr><td>B</td><td>Select the relative order of `ROM`, `RAM`, and `Execution efficiency` based on application requirements.</td></tr>
<tr><td>C</td><td>Configure the Code Generation Advisor to run before generating code by setting the **Check model before generating code** configuration parameter to `On (proceed with warnings)` or `On (stop for warnings)`.</td></tr>
<tr><td>Notes</td><td colspan="2">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.</td></tr>
<tr><td>Rationale</td><td>A, B, C</td><td>When you use the Code Generation Advisor, configuration parameters conform to the objectives that you want and they are consistently enforced.</td></tr>
<tr><td>See also</td><td colspan="2">• “Application Objectives Using Code Generation Advisor” (Simulink Coder)
• “Manage Configuration Sets for a Model”</td></tr>
<tr><td>Last Changed</td><td colspan="2">R2015b</td></tr>
</table>

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 these configuration parameters to `warning` or `error`: • **Single task rate transition** • **Enforce sample time specified by Signal Specification blocks** • **Detect multiple driving blocks executing at the same time step** For **multitasking** models, set these configuration parameters to `warning` or `error`: • **Multitask task rate transition** • **Multitask conditionally executed subsystem** • **Tasks with equal priority** If the model contains Data Store Memory blocks, set these configuration parameters to `Enable all as warnings` or `Enable all as errors`: • **Detect read before write** • **Detect write after read** • **Detect write after write** • **Multitask data store**
Rationale	Setting diagnostic configuration parameters improves run-time detection of rate and tasking errors.
See Also	• “Model Configuration Parameters: Diagnostics” • “hisl_0013: Usage of data store blocks” • “hisl_0044: Configuration Parameters > Diagnostics > Sample Time” • “hisl_0303: Configuration Parameters > Diagnostics > Data Validity > Merge blocks”
Last Changed	2016a

Component Deployment Using Service Interface Configuration

cgsl_0401: Modeling styles for component deployment

<table>
<tr><th>ID: Title</th><th colspan="2">cgsl_0401: Modeling styles for component deployment</th></tr>
<tr><td rowspan="3">Description</td><td colspan="2">A model intended for component deployment with a service interface shall be designed by using one of the following modeling styles:</td></tr>
<tr><td>A</td><td>Export-function modeling

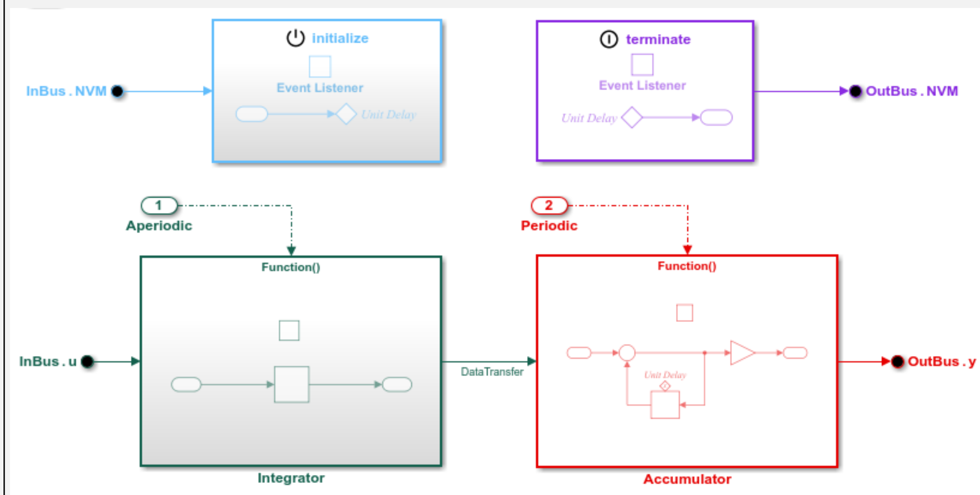
This modeling style supports single and multiple rates.</td></tr>
<tr><td>B</td><td>Rate-based modeling

This modeling style supports only a single rate.</td></tr>
<tr><td>Notes</td><td colspan="2">For export-function models, the code generator produces initialize and terminate entry-point functions and an entry-point function for each callable function represented in the model.

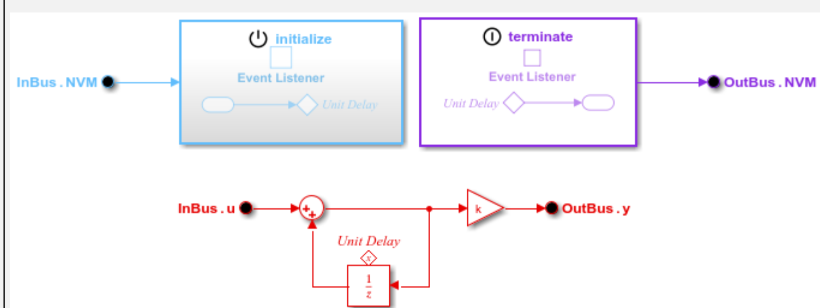
For rate-based models, the code generator produces initialize and terminate entry-point functions and an entry-point function for the rate of the model.</td></tr>
<tr><td>Rationale</td><td colspan="2">Support generation of callable entry-point functions in a component modeling architecture.</td></tr>
<tr><td>Model Advisor Check</td><td colspan="2">Verify this guideline by using Model Advisor check “Check modeling style for component deployment” (Embedded Coder)</td></tr>
</table>

<table>
<tr><th>ID: Title</th><th>cgsl_0401: Modeling styles for component deployment</th></tr>
<tr><td>Examples</td><td>

Export-Function Modeling Style



Rate-Based Modeling Style



</td></tr>
</table>

See Also

“Code Interfaces and Code Interface Specification” (Embedded Coder)

“Periodic and Aperiodic Function Interfaces” (Embedded Coder)

“Export-Function Models Overview”

“Rate-Based Models Overview”

“Create a Service Interface Configuration” (Embedded Coder)

“What Is Sample Time?”

“Specify Sample Time”

Version History

Introduced in R2022b

cgsl_0402: Signal interfaces for component deployment

<table>
<tr><th>ID: Title</th><th colspan="2">cgsl_0402: Signal interfaces for component deployment</th></tr>
<tr><td rowspan="3">Description</td><td colspan="2">At the root level of a component, signal interfaces shall be modeled by using only one type of signal:
• In Bus Element and Out Bus Element blocks
• Inport and Outport blocks</td></tr>
<tr><td>A</td><td>For structured signals that use In Bus Element and Out Bus Element blocks, set block parameters as follows:
• Data type to <code>Bus: <object name></code>.
• Bus virtuality to <code>nonvirtual</code>.
Configure the interface for each In Bus Element and Out Bus Element block individually.</td></tr>
<tr><td>B</td><td>For structured signals that use Inport and Outport blocks, set block parameters as follows:
• Data type to <code>Bus: <object name></code>.
• Specify that the outport bus is nonvirtual at the root level by selecting Outport block parameter Output as nonvirtual bus in parent model.
• Specify that the output for a top-level Inport block used to load bus data is nonvirtual by selecting Inport block parameter Output as nonvirtual bus.</td></tr>
<tr><td>Notes</td><td colspan="2">Do not use datastore memory for signal interfaces.</td></tr>
<tr><td>Rationale</td><td colspan="2">Reduces complexity and provides model clarity.</td></tr>
<tr><td>Model Advisor Check</td><td colspan="2">Verify this guideline by using Model Advisor check “Check signal interfaces” (Embedded Coder)</td></tr>
</table>

See Also

“Code Interfaces and Code Interface Specification” (Embedded Coder)

“Create Nonvirtual Buses”

“Specify Bus Properties with Simulink.Bus Object Data Types”

“Composite Interface Guidelines”

In Bus Element

Out Bus Element

Inport

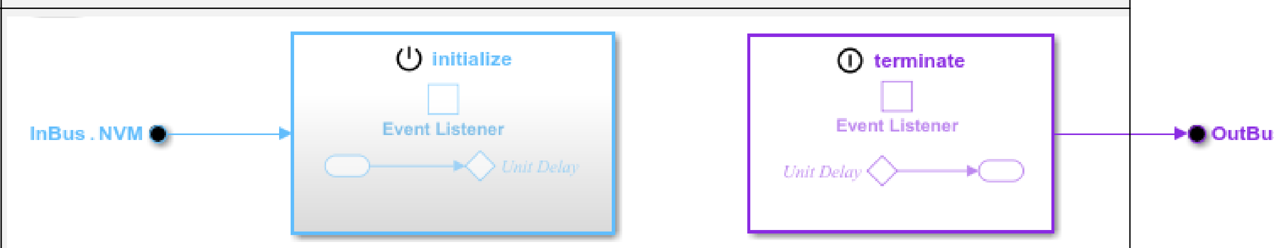
Outport

Version History

Introduced in R2022b

cgsl_0404: Model startup and shutdown events by using Initialize Function and Terminate Function blocks for component deployment

ID: Title	**cgsl_0404: Model startup and shutdown events by using Initialize Function and Terminate Function blocks for component deployment**
Description	To model startup and shutdown behavior, use Initialize Function and Terminate Function blocks
Notes	By following this guideline, the code generator produces one initialize function and one terminate function. When a Terminate Function block is not needed in the model, clear model configuration parameter **Terminate function required** (`IncludeMdlTerminateFcn`).
Rationale	Decouples the execution order of component initialize and terminate functions from the execution order across components. Separates component startup and shutdown functionality from periodic and aperiodic algorithm function code.
Model Advisor Check	A Model Advisor check is not provided for this guideline.

Examples	
	`void CD_initialize(void)`
	`{`
	`   .`
	`   .`
	`   .`
	`}`
	
	`void CD_terminate(void)`
	`{`
	`   .`
	`   .`
	`   .`
	`}`

See Also

“Startup, Reset, and Shutdown Function Interfaces” (Embedded Coder)

“Periodic and Aperiodic Function Interfaces” (Embedded Coder)

“Code Interfaces and Code Interface Specification” (Embedded Coder)

Initialize Function

Terminate Function

"Using Initialize, Reinitialize, Reset, and Terminate Functions"

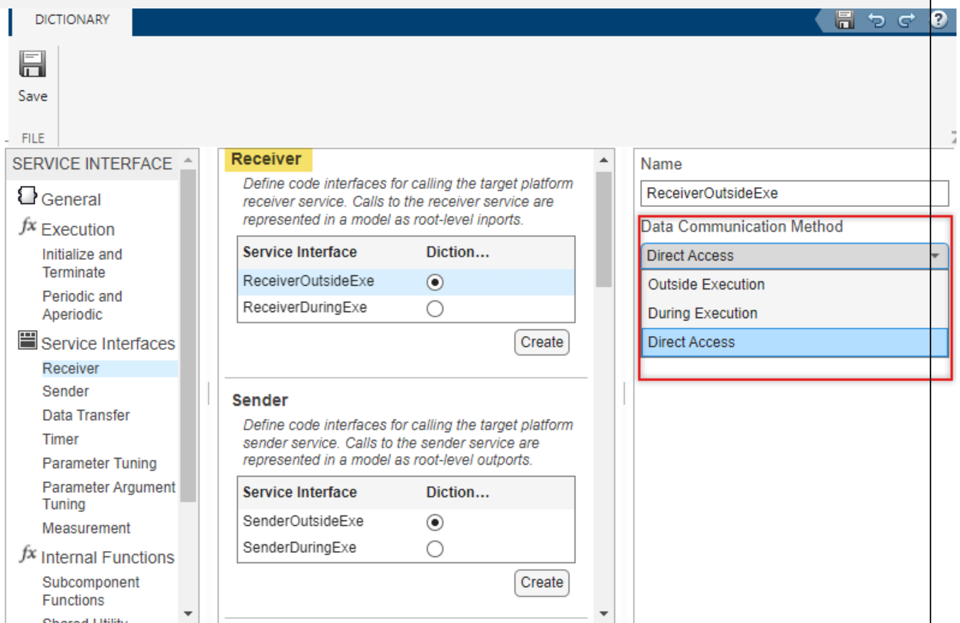
Version History

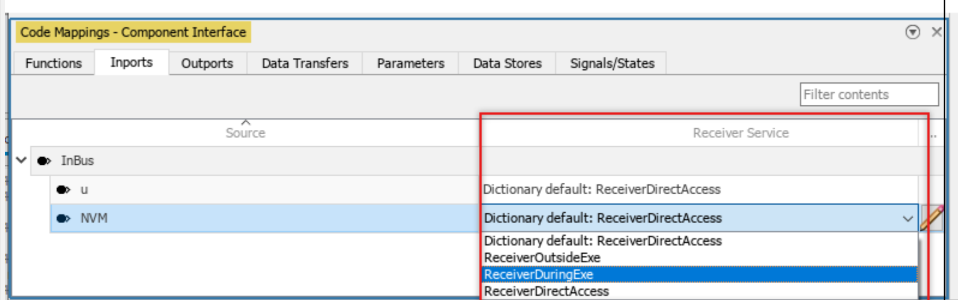
Introduced in R2022b

cgsl_0405: Data receive for component deployment

ID: Title	cgsl_0405: Data receive for component deployment	
Description	A	To model a call to the target platform receiver service, use an In Bus Element or Inport block.
	B	To safeguard data for concurrent access, map the component inports to a service interface that is configured to use the **During Execution** or **Outside Execution** communication method. • **During Execution** — The generated callable function that implements the algorithm safeguards data access for concurrency. • **Outside Execution** —The target platform service safeguards data access for concurrency.
	C	When concurrent access to data is not a concern, map component inports to a service interface that is configured to use the **Direct Access** communication method. In this case, no safeguard for data access is provided.
Rationale	The generated code aligns with the data communication method that is required by the target platform environment.	
Model Advisor Check	A Model Advisor check is not necessary for this guideline because a service interface for a receiver service must be configured to use one of the three data communication methods.	

ID: Title	cgsl_0405: Data receive for component deployment
Example	**Specifying the Data Communication Method for Calling the Target Platform Data Receiver Service**





In this example, the data communication method is set to **Outside Execution**.

```
void CD_integrator(void)
{
  .
  .
  .
  for (i = 0; i < 10; i++) {
    .
    .
    .
    rtDWork.DiscreteTimeIntegrator_PREV_U[i] = (get_CD_integrator_input())[i];
  }
  .
  .
  .
}
```

In this example, the data communication method is set to **During Execution**.

ID: Title	cgsl_0405: Data receive for component deployment

	<pre>void CD_integrator(void)
	{
	  .
	  .
	  .
	real_T tmp[10];
	  .
	  .
	  .
	  get_CD_integrator_input(&tmp[0]);
	  .
	  .
	  .
	  for (i = 0; i < 10; i++) {
	    .
	    .
	    .
	    rtDWork.DiscreteTimeIntegrator_PREV_U[i] = tmp[i]
	  }
	  .
	  .
	  .
	}</pre>In this example, the data communication method is set to **Direct Access**.<pre>void CD_integrator(void)
	{
	  .
	  .
	  .
	  for (i = 0; i < 10; i++) {
	    .
	    .
	    .
	    ... = CD_sig.In[i];
	  }
	  .
	  .
	  .
	}</pre>

See Also

“Code Interfaces and Code Interface Specification” (Embedded Coder)

“Receiver and Sender Service Interfaces” (Embedded Coder)

(Embedded Coder)

Embedded Coder Dictionary (Embedded Coder)

“Select Code Generation Output for Target Platform Deployment” (Embedded Coder)

In Bus Element block

Inport block

`get` (Embedded Coder) function

Version History

Introduced in R2022b

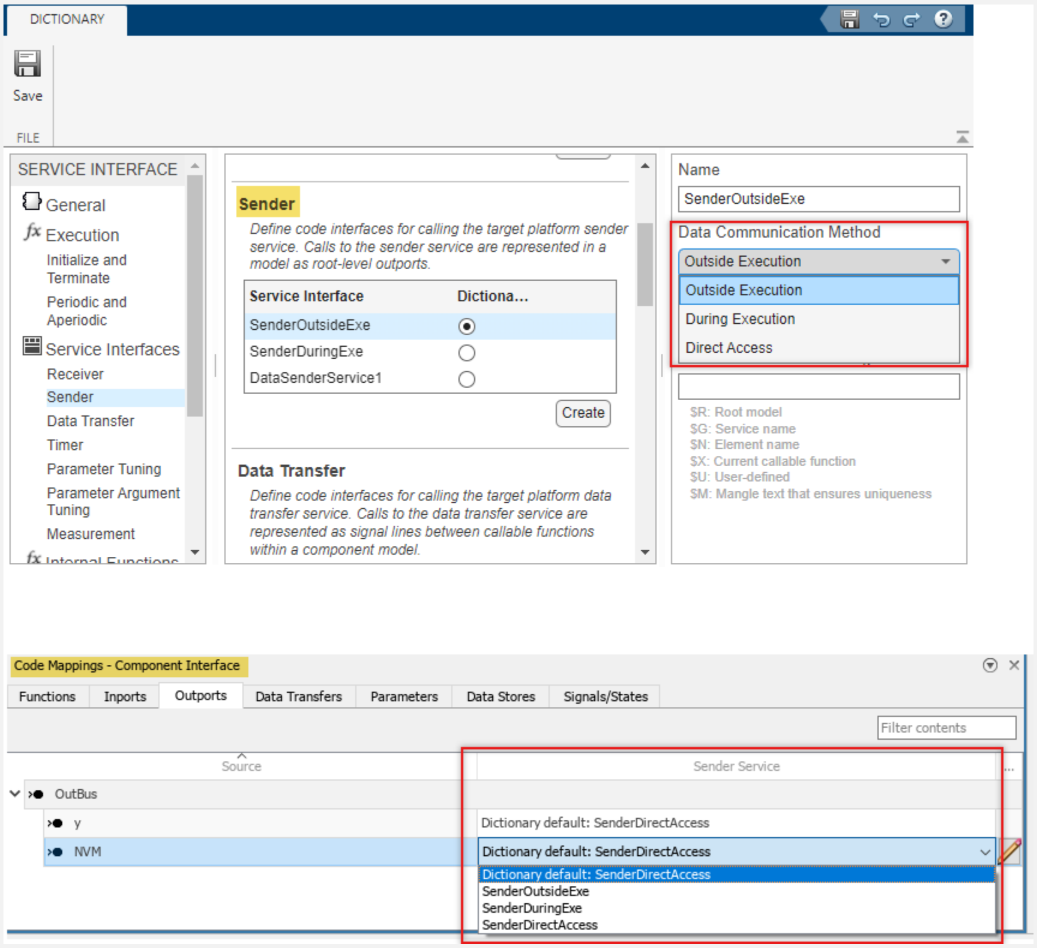
cgsl_0406: Data send for component deployment

<table>
<tr><th>ID: Title</th><th colspan="2">cgsl_0406: Data send for component deployment</th></tr>
<tr><td rowspan="3">Description</td><td>A</td><td>To model a call to the target platform sender service, use an Out Bus Element or Outport block.</td></tr>
<tr><td>B</td><td>To safeguard data for concurrent access, map the component outports to a service interface that is configured to use the During Execution or Outside execution communication method.

• During Execution —The generated callable function that implements the algorithm safeguards data access for concurrency.

• Outside Execution — The target platform service safeguards data access for concurrency.</td></tr>
<tr><td>C</td><td>When concurrent access to data is not a concern, map component outports to a service interface that is configured to use the Direct Access communication method. In this case, no safeguard for data access is provided.</td></tr>
<tr><td>Rationale</td><td colspan="2">The generated code aligns with the data communication method that is required by the target platform environment.</td></tr>
<tr><td>Model Advisor Check</td><td colspan="2">A Model Advisor check is not necessary for this guideline because a service interface for a sender service must be configured to use one of the three data communication methods.</td></tr>
</table>

ID: Title	cgsl_0406: Data send for component deployment
Example	**Specifying the Data Communication Method for Calling the Target Platform Data Sender Service**



In this example, the data communication method is set to **Outside Execution**.

```
void CD_accumulator(void)
{
  .
  .
  .
  for (i = 0; i < 10; i++) {
    .
    (set_CD_accumulator_out())[i] = CD_param.tunable_gain * CD_sig.delay[i];
  }
}
```

In this example, the data communication method is set to **During Execution**.

```
void CD_accumulator(void)
{
  real_T out[10];
  .
  .
  .
```

<table>
<tr><th>ID: Title</th><th>cgsl_0406: Data send for component deployment</th></tr>
<tr><td></td><td>

```
  for (i = 0; i < 10; i++) {
    .
    out[i] = CD_param.tunable_gain * CD_sig.delay[i];
  }
  set_CD_accumulator_out(&out[0]);
}
```

In this example, the data communication method is set to **Direct Access**.

```
void CD_accumulator(void)
{
  .
  .
  .
  for (i = 0; i < 10; i++) {
    .
    .
    .
    CD_sig.out[i] = CD_param.tunable_gain * CD_sig.delay[i];
  }
}
```

</td></tr>
</table>

See Also

"Code Interfaces and Code Interface Specification" (Embedded Coder)

"Receiver and Sender Service Interfaces" (Embedded Coder)

"Data Communication Methods" (Embedded Coder)

Embedded Coder Dictionary (Embedded Coder)

"Select Code Generation Output for Target Platform Deployment" (Embedded Coder)

Out Bus Element

Outport

`set` (Embedded Coder)

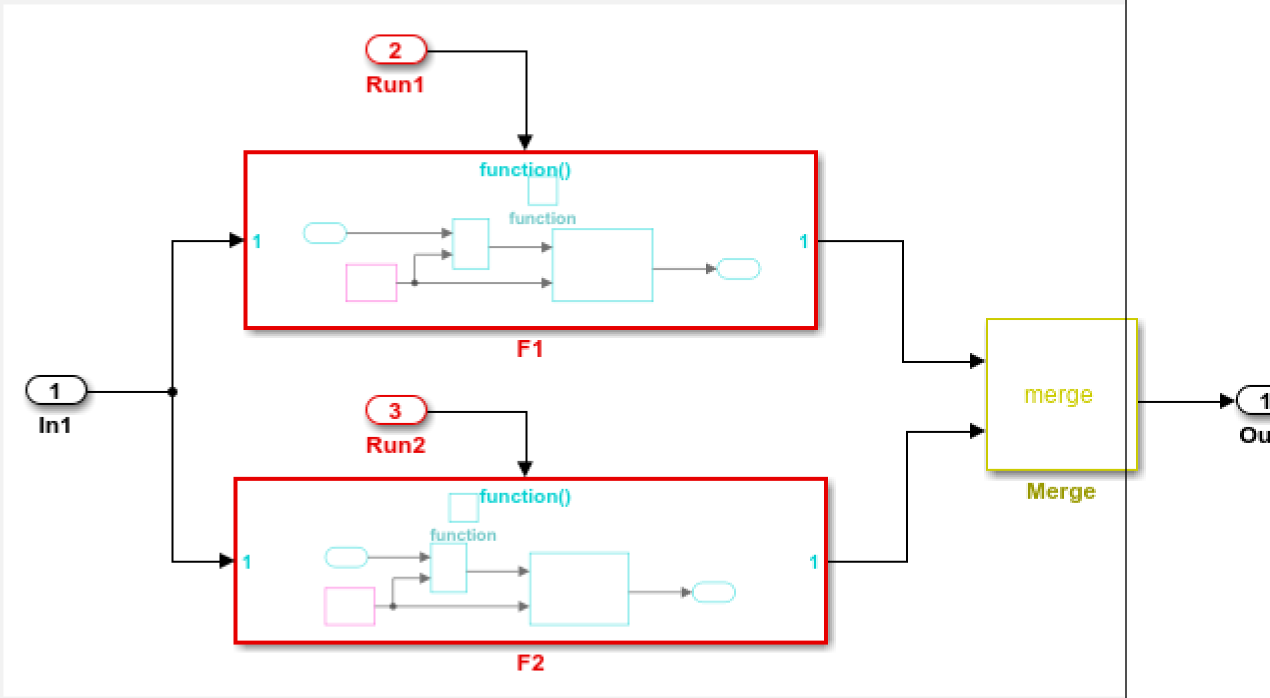
Version History

Introduced in R2022b

cgsl_0408: Partial data send for component deployment

ID: Title	**cgsl_0408: Partial data send for component deployment**	
Description	To model a partial data send, set the data communication method to **Direct Access** and:	
	A	Use an Assignment block to model mutually-exclusive partial write operations.
	B	Use a Merge block when writing data from multiple functions.
	C	Configure the outports on the signal path of the component root-level outport for the partial data send as virtual. To do so, select Outport block parameter **Ensure outport is virtual**.
	D	Map the root-level outport for the partial data send to a service interface that is configured for direct-access data communication. The signal data is not safeguarded for concurrent access.
Notes	This guideline is only applicable for the export-function modeling style. An Out Bus Element block cannot be used when modeling partial data write.	
Rationale	Promotes efficient code by avoiding data copies.	
Model Advisor Check	A Model Advisor check is not provided for this guideline.	

ID: Title	cgsl_0408: Partial data send for component deployment

Examples	
	void Run1(void)
	{
	   Out[1] = In + 1.0;
	}
	
	void Run2(void)
	{
	   Out[0] = In;
	}

See Also

“Code Interfaces and Code Interface Specification” (Embedded Coder)

“Service Interfaces” (Embedded Coder)

“Data Communication Methods” (Embedded Coder)

Embedded Coder Dictionary (Embedded Coder)

“Target Environment Services” (Embedded Coder)

Assignment

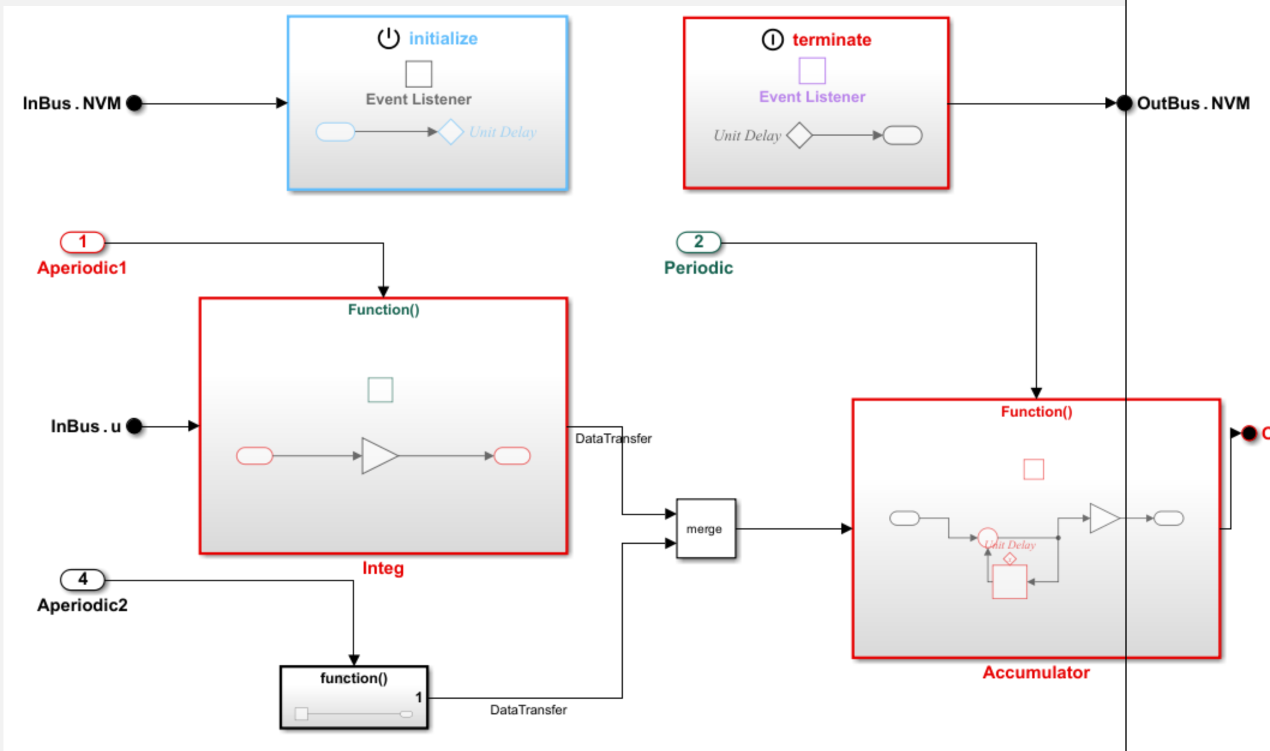
“Ensure Outport is Virtual”

Version History

Introduced in R2022b

cgsl_0409: Data transfer for component deployment

ID: Title	cgsl_0409: Data transfer for component deployment	
Description	To model data transfers:	
	A	Use signals between callable functions.
	B	When branching or merging transfer signals, in the Embedded Coder dictionary, add $X to the **Function Naming Rule** fields. Compliance with this rule is enforced during code generation.
	C	Do not branch data transfer signals to the root-level output port. Compliance with this rule is enforced during code generation.
Notes	When merging data transfer signals, ensure that both signals are mutually exclusive.	
Rationale	The generated code aligns with the data communication method required by the platform environment for concurrent execution.	
Model Advisor Check	A Model Advisor check is not provided for this guideline.	

ID: Title	cgsl_0409: Data transfer for component deployment
Examples	


```
void CD_accumulator(void)
.
.
.
  tmpIrvIRead = get_CD_accumulator_DataTransfer();
.
.
.

void CD_integrator(void)
.
.
.
  tmp = set_CD_integrator_DataTransfer();
.
.
.

void CD_Aperiodic2(void)
.
.
  tmp = set_CD_Aperiodic2_DataTransfer();
.
.
.
```

See Also

"Code Interfaces and Code Interface Specification" (Embedded Coder)

"Data Transfer Service Interfaces" (Embedded Coder)

"Data Communication Methods" (Embedded Coder)

Embedded Coder Dictionary (Embedded Coder)

"Target Environment Services" (Embedded Coder)

"Select Code Generation Output for Target Platform Deployment" (Embedded Coder)

"Configure Signal Data for C Code Generation" (Embedded Coder)

`get` (Embedded Coder)

`set` (Embedded Coder)

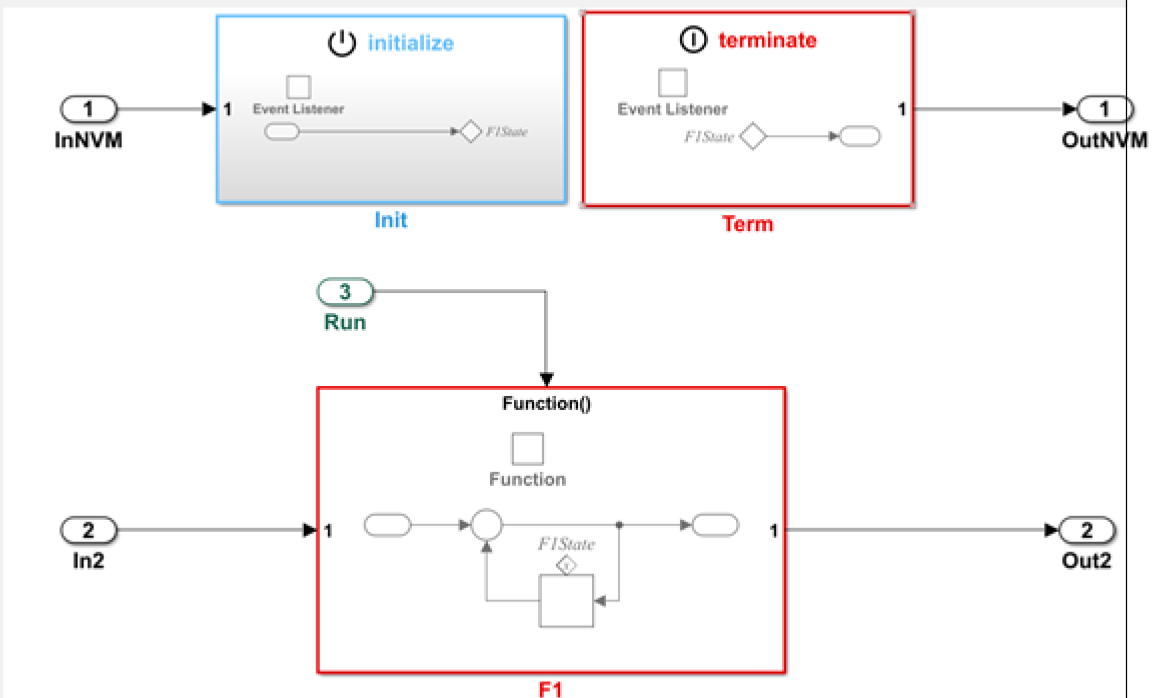
Version History

Introduced in R2022b

cgsl_0411: Access nonvolatile memory by using Initialize Function and Terminate Function blocks

<table>
<tr><th>ID: Title</th><th colspan="2">0411: Access nonvolatile memory by using Initialize Function and Terminate Function blocks</th></tr>
<tr><td rowspan="3">Description</td><td colspan="2">To model the Direct Access data communication method to target platform nonvolatile memory:</td></tr>
<tr><td>A</td><td>At the root-level of the component, use the Initialize Function block to read data and the Terminate Function block to write data.</td></tr>
<tr><td>B</td><td>Configure the root-level ports to use the Direct Access data communication method.</td></tr>
<tr><td>Notes</td><td colspan="2">When accessing nonvolatile memory during function execution, see guideline “cgsl_0406: Data send for component deployment” on page 5-13 and “cgsl_0405: Data receive for component deployment” on page 5-9.

When you need to access nonvolatile memory by using a service provided by the target environment, use a client-server interface approach for modeling the interface. With that approach you represent the target environment service that provides access to nonvolatile memory by using a Simulink Function block and access the service by using the Function Caller block. For more information, see “Nonvolatile Memory Interfaces” (Embedded Coder).</td></tr>
<tr><td>Rationale</td><td colspan="2">• Robust handling of data access by functions that execute concurrently.
• Supports multiple instances of components.</td></tr>
<tr><td>Model Advisor Check</td><td colspan="2">A Model Advisor check is not provided for this guideline.</td></tr>
</table>

<table>
<tr><th>ID: Title</th><th>0411: Access nonvolatile memory by using Initialize Function and Terminate Function blocks</th></tr>
<tr><td>Examples</td><td>


<pre>
void CD_initialize(void)
.
.
.
    &(get_CD_initialize_input())[0]
.
.
.
void CD_terminate(void)
{
  memcpy[&(getref_CD_terminate_OutBus_NVM())[0]...
}
</pre>
</td></tr>
</table>

See Also

“Code Interfaces and Code Interface Specification” (Embedded Coder)

“Service Interfaces” (Embedded Coder)

“Client-Server Interface” (Embedded Coder)

Initialize Function

Terminate Function

Reset Function

“Using Initialize, Reinitialize, Reset, and Terminate Functions”

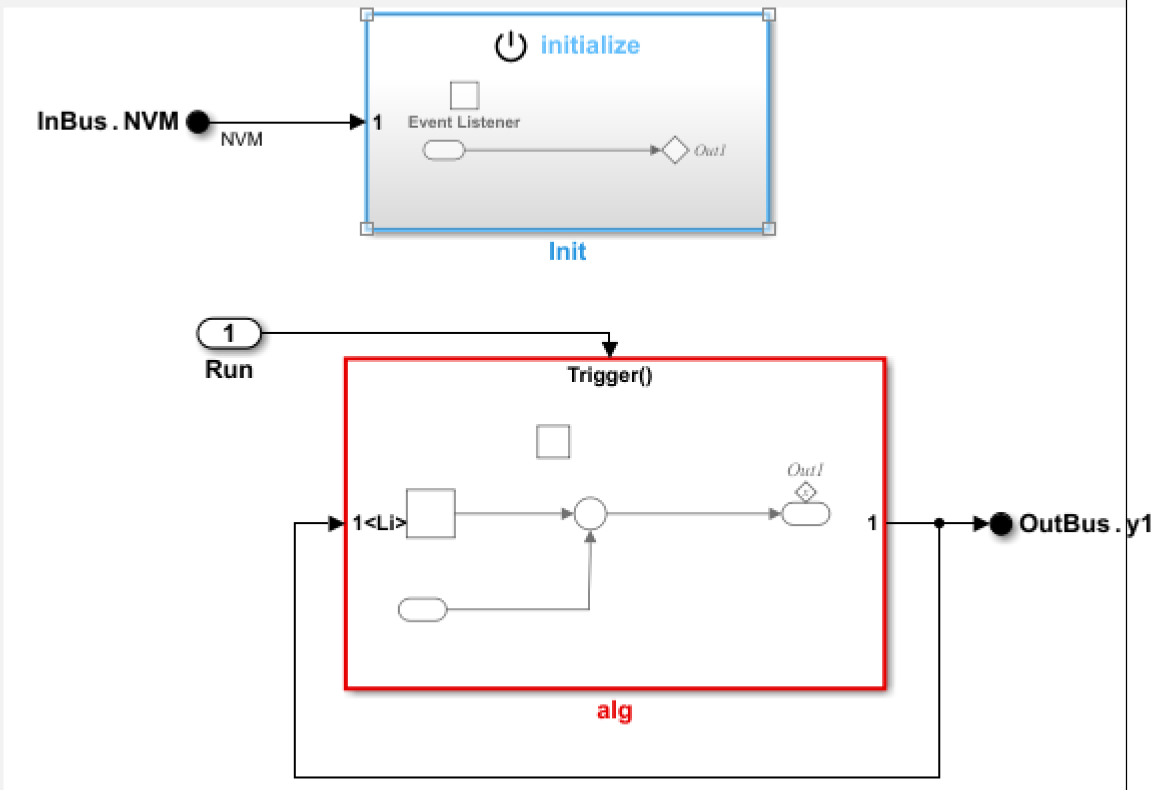
Version History

Introduced in R2022b

cgsl_0413: Reuse memory between component state and output for component deployment

ID: Title	cgsl_0413: Reuse memory between component state and output for component deployment	
Description	To optimize component memory usage by reusing memory for state and output data, use one of these methods:	
	A	Use a function loopback pattern to model the state variable as a signal.
	B	Use a Delay block to model the state variable explicitly. Set the state of the Delay block and the function output port to the same literal initial condition value.
Notes	This approach is applicable for data communication methods **Outside Execution** and **Direct Access** because these methods can access persistent memory. For method B, the code generator makes a best effort to optimize memory usage. Under some conditions, such as when initialization is done dynamically by using a signal rather than a parameter, the code generator might not apply the optimization. If the optimization does not occur, consider using method A. Regardless of whether you use approach A or B, the code generator implements robust handling of data access by functions that execute concurrently.	
Rationale	A	Reuse of memory for state and output data. Optimization survives dynamic initialization.
	B	Reuse of memory for state and output data.
Model Advisor Check	A Model Advisor check is not provided for this guideline.	

<table>
<tr><th>ID: Title</th><th>cgsl_0413: Reuse memory between component state and output for component deployment</th></tr>
<tr><td>Examples</td><td>



In this example, the data communication method is set to "Direct Access".

```
void CD_accumulator(void)
{
   int32_T i;
   for (i=0; i<10; i++) {
     Out[i] += In[i];
   }
}
```

In this example, the data communication method is set to "Outside Execution".

```
void CD_accumulator(void)
{
   real_T tmpIrvRead[10];
   int32_T i;
   tmp = set_CD_accumulator_her_out_y();
   for (i=0; i<10; i++) {
     tmp[i] = (get_CD_accumulator_DataTransfer(tmpIrvRead))[i] + tmp[i];
   }
}
```

</td></tr>
</table>

See Also

"Code Interfaces and Code Interface Specification" (Embedded Coder)

“Service Interfaces” (Embedded Coder)

Embedded Coder Dictionary (Embedded Coder)

“Data Communication Methods” (Embedded Coder)

Delay

“Use dynamic memory allocation for model initialization” (Embedded Coder)

Version History

Introduced in R2022b

cgsl_0414: Configure service interface for component model

<table>
<tr><th>ID: Title</th><th colspan="2">cgsl_0414: Configure service interface for component model</th></tr>
<tr><td rowspan="4">Description</td><td colspan="2">The following configuration shall be applied:</td></tr>
<tr><td>A</td><td>Link the model to an Embedded Coder dictionary that defines a service code interface.</td></tr>
<tr><td>B</td><td>Configure deployment types as:
• Component for the top model
• Subcomponent for the submodel (referenced model)</td></tr>
<tr><td>C</td><td>Use the Code Mappings editor or code mappings programming interface to map model elements that represent interfaces to service interfaces that are defined in the linked coder dictionary.</td></tr>
<tr><td>Rationale</td><td colspan="2">Deploy models as components that include comprehensive service interface support, including support for concurrent data access.

Generate component model code intended to interact with service implementations of a target platform.</td></tr>
<tr><td>Model Advisor Check</td><td colspan="2">Verify this guideline by using Model Advisor check “Check configuration for component deployment” (Embedded Coder)</td></tr>
</table>

See Also

“Code Interfaces and Code Interface Specification” (Embedded Coder)

“Create a Service Interface Configuration” (Embedded Coder)

Embedded Coder Dictionary (Embedded Coder)

Version History

Introduced in R2022b