Two Methods for Breaking Data Dependency Loops in System Level Models

June 20, 2007
Version 1.0
Author: Michael Burke
The MathWorks™

Table of Contents

Introduction 3
Task Overview 3
Assumptions 4
Preparation 5
Select an Appropriately Sized Model 5
Determine "Desired" Order of Execution 5
Resolve Rate Transitions between Subsystems Running at Different Rates 6
Using Modeling Styles to Simplify Visualization of the Model 7
Implementation 8
Function--Call Method 8
Manual Control of Execution Order 9
Setting the Desired Execution Order 9
Resolve Remaining Data Dependencies 11
Remove Unnecessary Unit Delays 19
Mixed Function Call and Manual Control 20
Model Analysis 20
Verification of Execution Order for Manual Control Method 20
Verification of Execution Order for Function Call Driven Method 21
Appendix 23
Understanding Execution Order 23
How Simulink Determines the Sorted Order 23
About Direct-Feed through Ports 23
Data Dependency 23
Basic Algebraic Loops 24
The Affect of Breaking Data Dependencies on System Behavior 25
Data Independent Subsystems 27
Simulink Sorted Order 28
Function Call Initiators 29
Atomic Subsystems and Execution Order 30
Referenced MAAB Rules 32
jc_0171: Maintaining signal flow when using Goto and From blocks 32
db_0143: Similar block types on the model levels 33

Introduction

Creation of a system by integrating multiple validated subcomponents is a task common to Simulink® and C. In both environments integration issues arise as the number of components increases. Problems include miss-matched data types, scaling and ensuring the correct signals are available, to name a few. These are all challenges that systems integrators are used to solving. However; the integration process in Simulink exposes **data dependency** issues that are normally hidden in the C development environment.

Data dependency is the requirement that for any calculation, all the values on the right hand side (RHS) of equation are known prior to starting the calculation. The value on the left hand (LHS) side is **dependent** on the values of the right hand side. In the contexts of subsystems, the concept of calculation order is equivalent to execution order of the subsystems.

$$LHS_n = f\,(RHS_n)$$

The C language does not prevent users from writing equations where the LHS is assigned before the RHS. This means that an old or non-initialized data can be used which can result in unexpected or incorrect results.

$$LHS_n = f\,(RHS_{n-1})$$
$$LHS_n = f\,(??)$$

Unlike C, Simulink contains built in analysis tools that prevent this from happening. Users of Simulink are familiar with these diagnostic, which are most commonly referred to as algebraic loops. Loops are broken by explicitly setting the order of calculation. This paper focuses on how to easily and systematically break these loops.

Two methods are covered in this paper, using function-call subsystems and unit delay blocks. Both of these methods can be used to define the execution order of the system.

Task Overview

The paper breaks the process into three tasks, preparation, implementation and analysis. The preparation phase places your model into a state which makes it easier to resolve the data dependency issues.

This paper covers two approaches for the iterative phase; a function call driven approach and the use of unit delays to break data dependencies. The analysis phase provides methods for insuring that the desired execution order is achieved.

Assumptions

This paper makes several assumptions regarding the state of a model, all of which do not have to be met for you to use this paper. The assumptions will simplify the process.

- The model consists of multiple subcomponents.
    - The subcomponents are atomic subsystems
    - No transformational blocks exist outside of the sub-components
      See MAAB Style Guide rule db_0143

- The model can use multiple rates provided
    - The subcomponents are single rate
    - A rate monotonic scheduler is used; faster rate subsystems run before slower rate subsystems

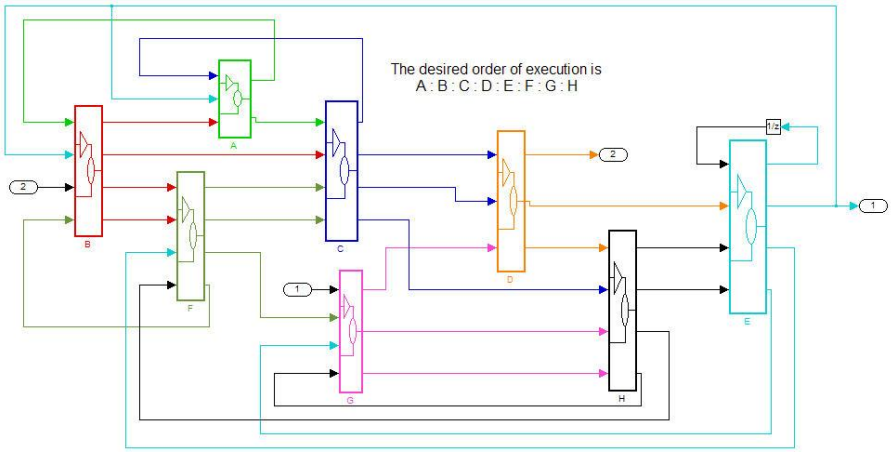
Preparation

- Select an appropriately sized model
- Determining the desired order of execution
- Resolving rate transitions between components running at different rates
- Using modeling styles to simplify visualization of the model

Select an Appropriately Sized Model

Since a full control algorithm may consist of hundreds or even thousands of functions, the first step is to select an appropriate sized model. A best practice is to design models with between 10 to 20 atomic subsystems at the top level. You can build up a full component, or full system, in turn is built up from a series of these smaller components. This approach of building larger models from a series of small components is consistent with C programming practices of building systems out of functions and groupings of functions.

This paper refers to the model shown below, `Control_Exe_Base.mdl`. Subsystems are colored to allow for easier visualization of the loops between systems.



Determine "Desired" Order of Execution

In some cases, there is a desired order of execution for the component. The desired order of execution could reflect the requirements of an existing external scheduler or it could be due to the physical reality of the system or the requirements of the underlying control laws. In either case, prior to modifying the model, the user should create a table that maps the desired relative order of execution of the atomic model's subsystems.

For example, a model, comprised of eight subsystems named A-H. Based on the design, there are constraints on the order of subsystem execution. The design might require an explicit execution order for the subsystems or it might require a relative order of execution. You can express relative constraints in terms of "Runs before" and "Runs after" as shown below.

Subsystem	Runs Before	Runs After
A	C,D	B

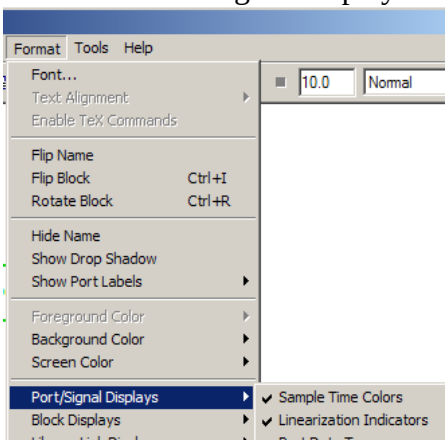
B	A,C	None
C	D,E	A,B
E	None	D

From the table, you can derive an explicit order of B,A,C,D,E. The subsystems F,G, and H do not have any user-imposed constraints. The execution order can be resolved based on the data dependency.

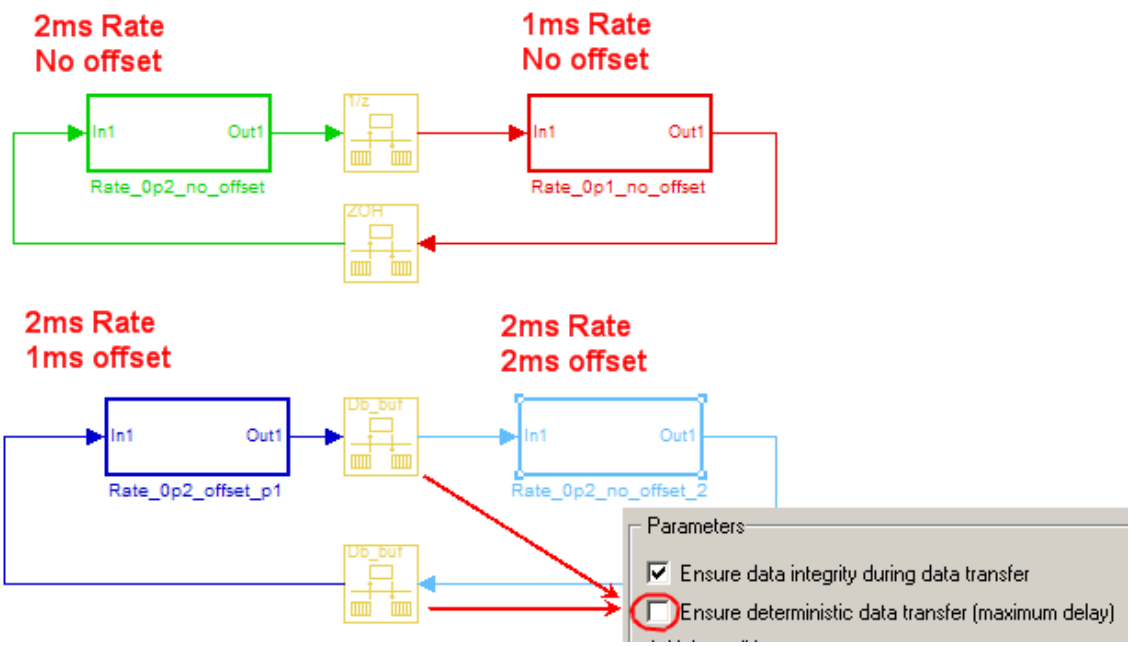
Resolve Rate Transitions between Subsystems Running at Different Rates

If the model uses multiple rates, the next step is to resolve rate transitions between subsystems. You can do this by using rate transition blocks.

1. Turn on the sample time highlighting
    a. Format > Port / Signal Display > Sample Time colors



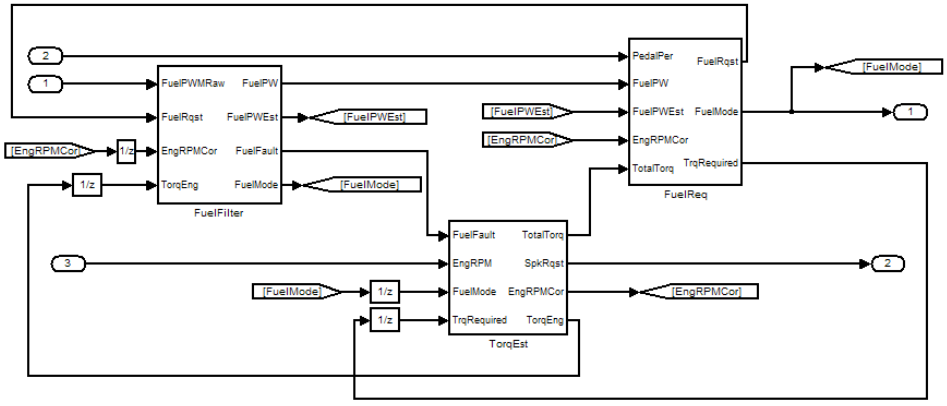
2. Add Rate Transition blocks between subsystems running at different rates
    a. Slow to fast transitions
    b. Fast to slow transitions
    c. Disable the block parameter "**Ensure deterministic rate transfer**" between subsystems that run at non integer multiples rates.



The MATLAB™ / Simulink documentation can be referenced for more information on how rate transition blocks function

Using Modeling Styles to Simplify Visualization of the Model

A clean layout simplifies working with a model. One of the most common layouts is the "left to right layout". In left to right layout, the signal flow starts on the left side of the diagram and flows to the right. Signals flowing right-to- left indicate a feedback loop. Goto and From blocks can be used to simplify the diagram layout, following the MAAB rule jc_0171.



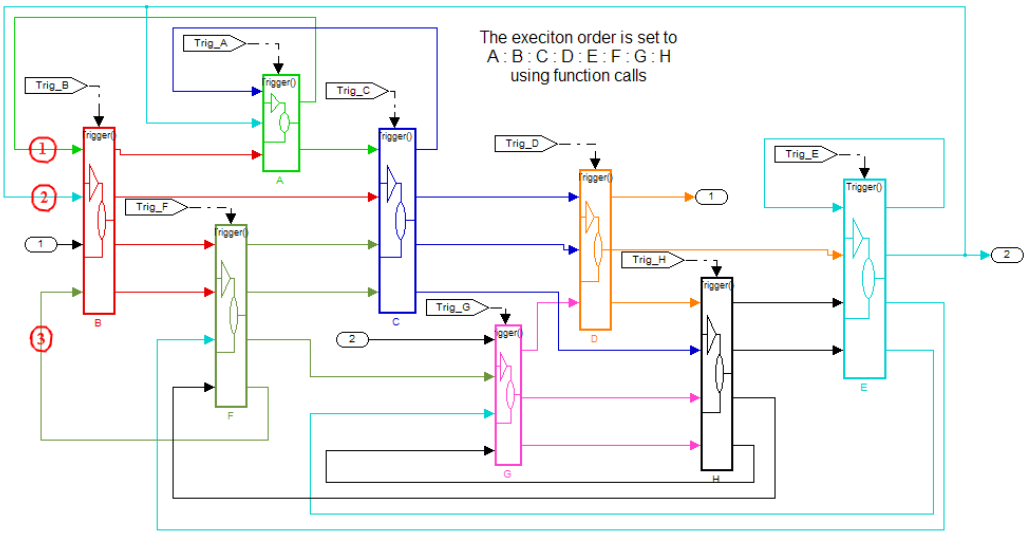
Implementation

This paper supports two methods for resolving data dependencies; function call driven and manual insertion. Both methods will resolve the data dependencies, however the two methods can, in some cases, result in different simulation behavior. For both methods the code generation and the simulated behavior will match.

	Pros	**Cons**
Function call	• Directly maps onto traditional schedulers • Direct explicit control of execution order • Any execution order can be set	• Requires additional scheduling subsystems
Unit Delay	• Uses Simulink data dependency algorithms to minimize data delays • Does not require additional scheduling subsystems / architecture	• Not all execution orders can be set • Changes to the model may require rearchitecting the unit delay configuration

Function--Call Method

The function-call method uses function-call initiators to explicitly set the order of execution. Since the order of execution is set all data dependencies are resolved.



For the function-call method to work best, adhere to the basic rules:

- Trigger a subsystem from only one function-call initiator.
  - You can trigger subsystems with more then one function call signal, provided the signals come from a common function-call initiator.

- For multiple rate do the following:
    - Use a single Stateflow diagram to mimic a multiple rate system
    - Use multiple function-call initiators
    - There should only be one function call initiator per rate and offset
    - Rate Transition blocks exist between blocks running at different rates.

Manual Control of Execution Order

You can control the execution order by using Unit Delay blocks. Unit Delay blocks maintain state information which breaks the downstream data dependency. If the user starts with a clean model, i.e. there are no Unit Delay blocks between the atomic subsystems, then only the first two steps of the process are required.

1. Set the desired execution order
2. Resolve data dependencies
3. Remove unnecessary Unit Delays

Setting the Desired Execution Order

The first step is to enforce the user desired execution order. The following steps will configure the model with the desired execution order. If two subsystems are data independent, e.g. there are no signals directly exchanged, Unit Delays can not be used to control their relative execution order.

If all the subsystems, in the model are included in the ordering routine this last step in the process.

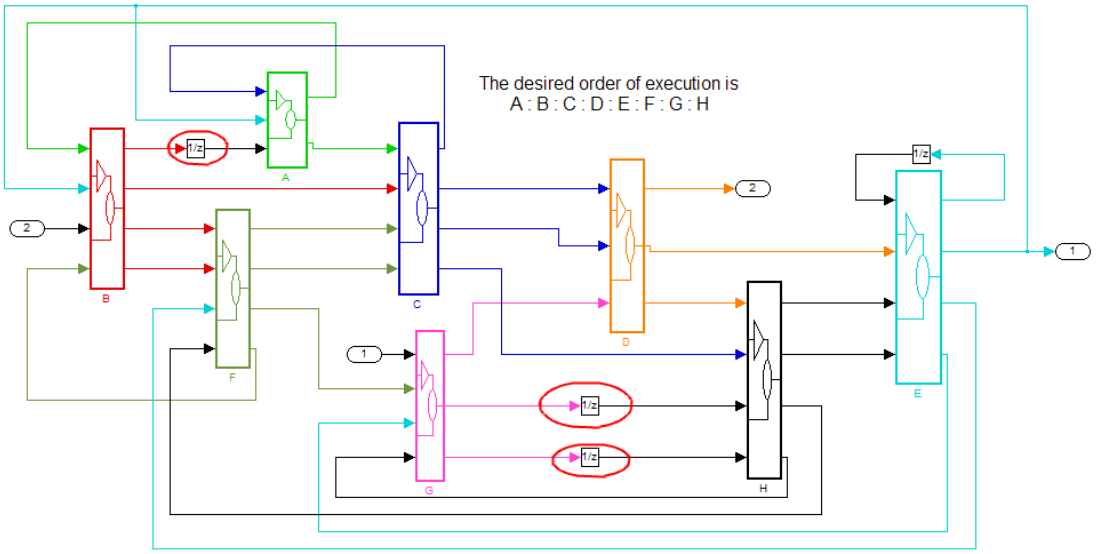
Step 1: Starting with the first subsystem and proceeding to the last, add Unit Delay blocks to the inputs for the N^{th} subsystem from the N^{th}+1 subsystem.

- All inputs to A from B should have a Unit Delay blocks.
- All inputs to B from C should have a Unit Delay blocks.

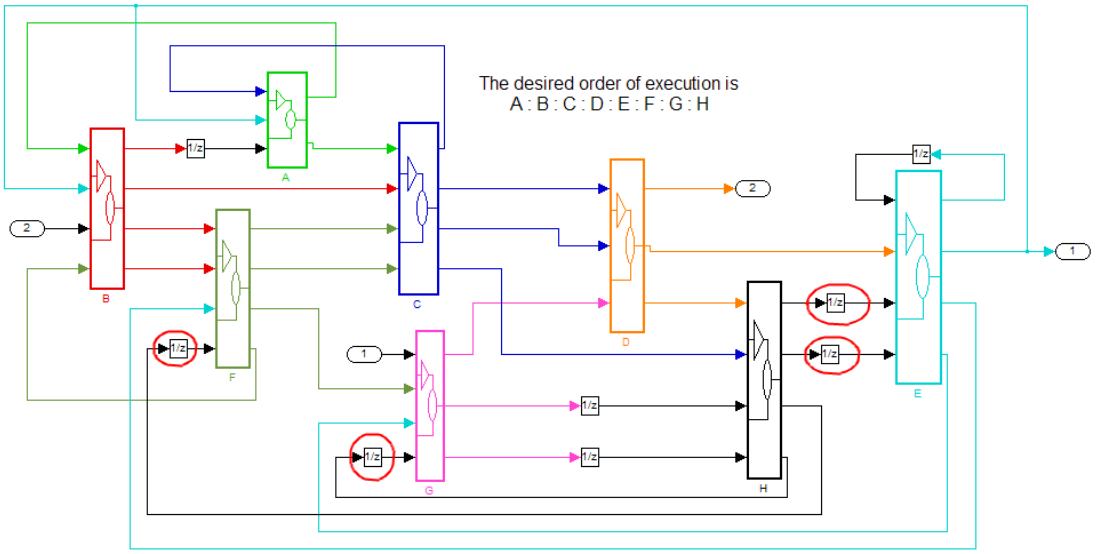
Step 2: Starting with the last subsystem add Unit Delays to all it's outputs when they used by "earlier subsystems". Perform an update diagram you add the Unit Delay blocks.

- All outputs from H going to subsystem A→G should have a Unit Delay
- All outputs from G going to subsystem A→F should have a Unit Delay

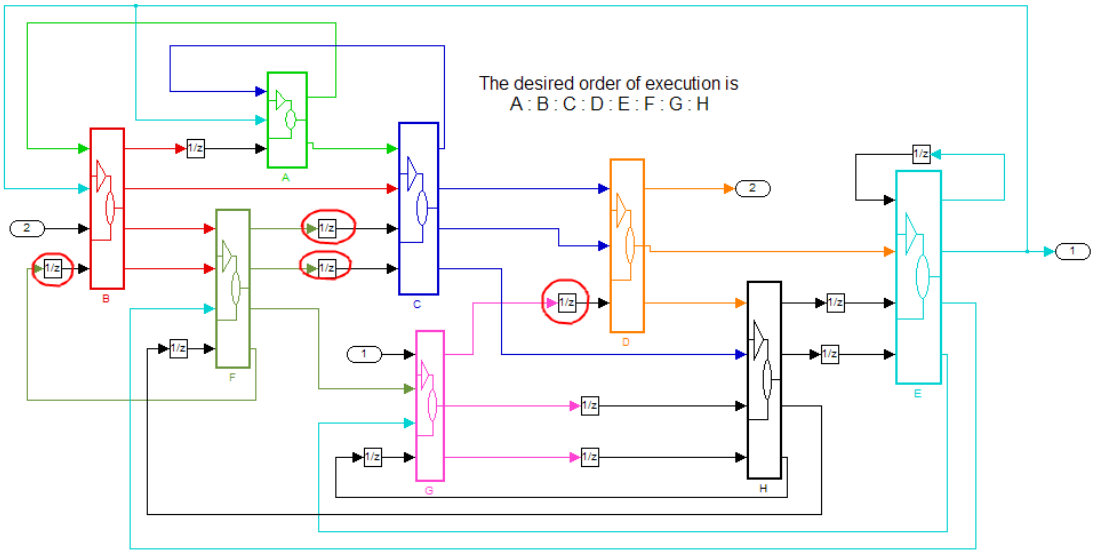
Example: The model Control_Exe_Base.mdl is used for in this example
For these steps, assume you have the following model with eight subsystems, A to H, and the desired order of execution is A→B→C→D→E→F→G→H
Step 1: Breaking signal flow between direct upstream subsystems (B→ A and G → H)



Step 2: 1st pass: Feedback from H

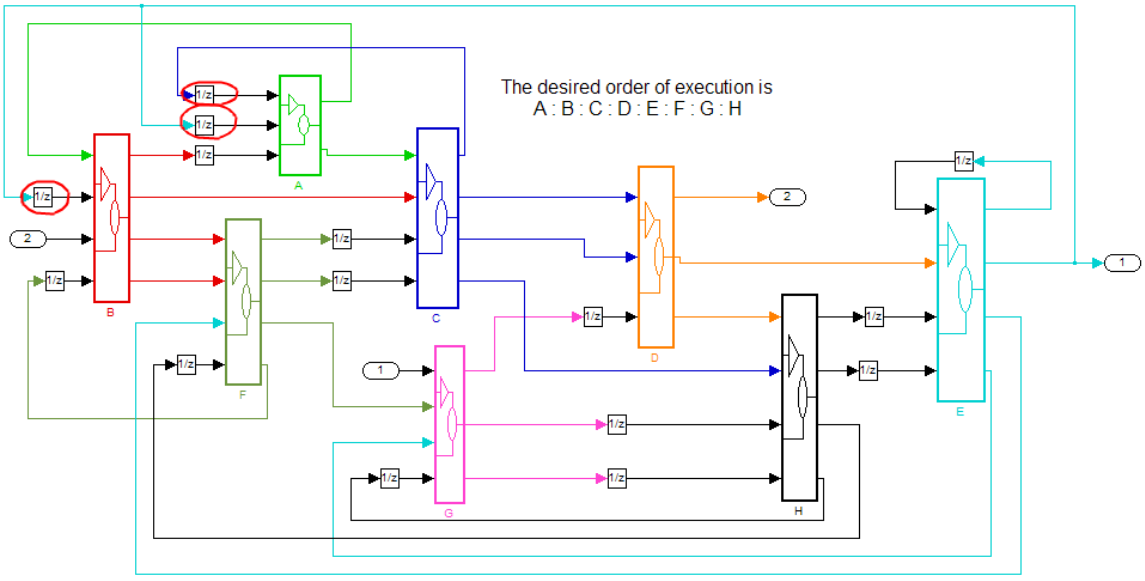


Step 2: 2nd & 3rd passes: (G & F)
Note: Since G runs after F, there is a unit delay



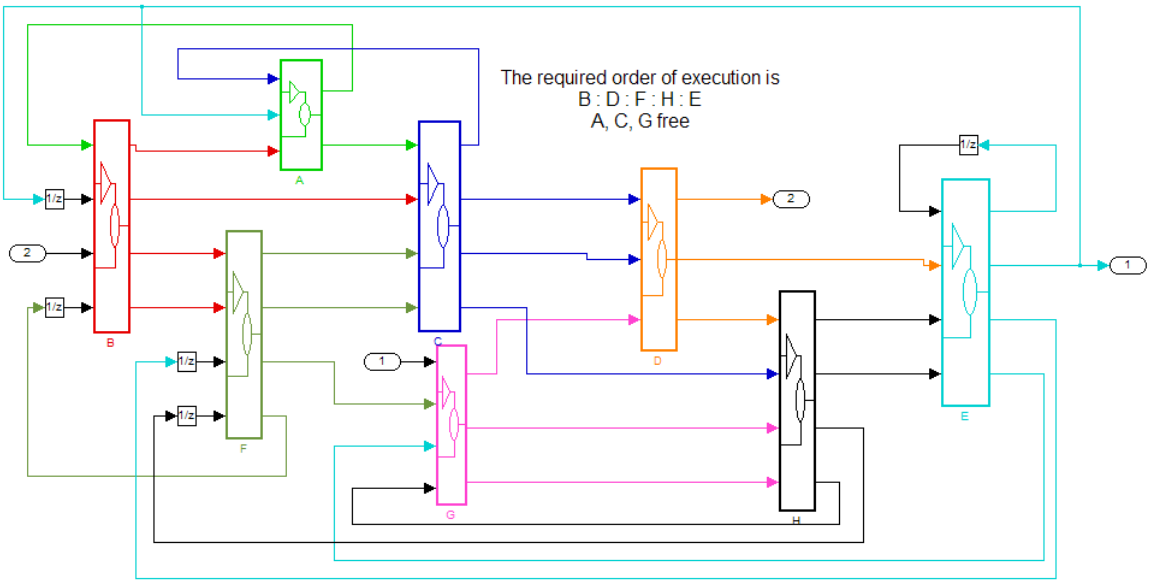
Step 2: 4th, 5th, 6th passes (E, D, C)

Note: D does not feed any earlier subsystems. Therefore, it has no unit delays added. After subsystem C, the models data dependencies are fully resolved and the desired execution order has been set.



Resolve Remaining Data Dependencies

In the example above, every subsystem was included in the order of execution. If the execution order was only specified for B, D, F, H and E then A, C and G would be considered "free". This could happen when the user only has requirements for a subset of the model and wants to optimize the remaining execution order. However, this will result in a model with algebraic loops.



There are two basic approaches to solving these loops. Arbitrarily specify an order of execution for the remaining subsystem. Then repeat the steps in phase 1, this time with the "free" subsystems controlled. To avoid the inclusion of unnecessary unit delays, start the process from a clean model without any Unit Delay blocks.

	Message	Source	Reported by	Summary
●	Block error	H	Simulink	Cannot solve algebraic loop involving '...
●	Block error	D	Simulink	Algebraic loop error with 'Control_Exe_...
●	Block error	C	Simulink	Algebraic loop error with 'Control_Exe_...
●	Block error	A	Simulink	Algebraic loop error with 'Control_Exe_...
●	Block error	E	Simulink	Algebraic loop error with 'Control_Exe_...
●	Block error	B	Simulink	Algebraic loop error with 'Control_Exe_...
●	Block error	F	Simulink	Algebraic loop error with 'Control_Exe_...
●	Block error	G	Simulink	Algebraic loop error with 'Control_Exe_...
●	Block error	H	Simulink	Algebraic loop error with 'Control_Exe_...

Control_Exe_Partial_Control_Manual/H

Cannot solve algebraic loop involving 'Control_Exe_Partial_Control_Manual/H' because it consists of blocks that cannot be assigned algebraic variables, i.e., blocks with discrete-valued outputs, blocks with non-double or complex outputs, Stateflow blocks, or nonvirtual subsystems.

- Resolve the data dependencies based on the error messages provided by Simulink. Following this approach will result in a system with the fewest unit delays.

Types of Error Messages

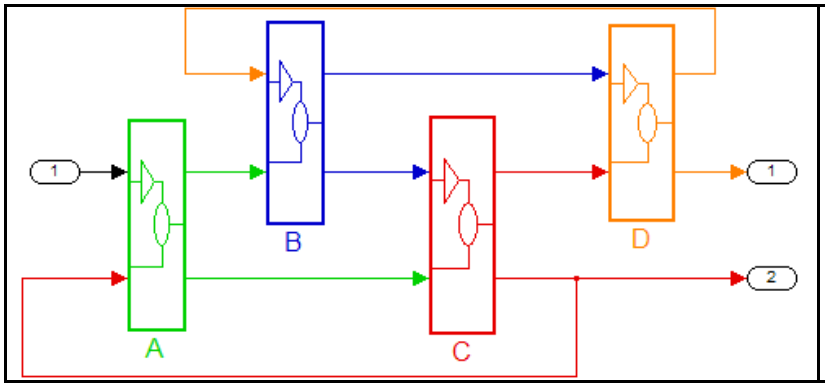
When Simulink finds a data dependency in a model it returns an error message. The error messages have two forms *a direct backwards loop* and *multiple co-existing*. The distinguishing characteristic of a direct backward loop is that when you click on the error messages in order, the signal flow between subsystems is direct and constant. In multiple

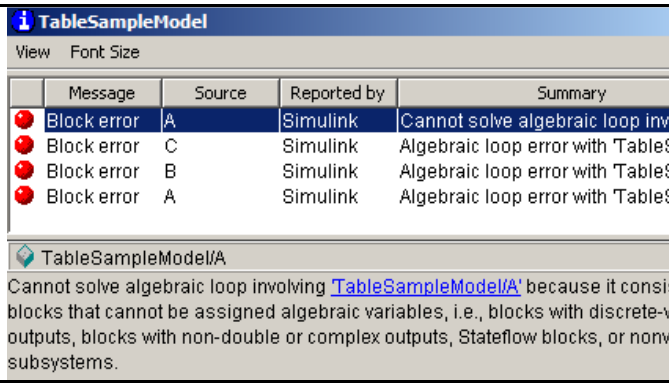
coexisting loops, the order of the error message is "broken." At some point in the error message, the block order either jumps without a direct connection between the subsystems or the connecting signals are broken by a unit delay. Both types of error message can be used to resolve model data dependencies.

	Direct backwards	**Multiple co-existing**
Number of loops in system	1	2 or more
Data loop can be directly traced from error message	Yes	No: error message "jump" at location of extra loops
Can be broken at a single point between two subsystems	Yes	No
Requires multiple iterations to resolve loops	No	Yes

Building a Connection Table

Starting with a simple model, we will look at how error messages can be used to break algebraic loops. We will examine the error messages using a *connection table*. The connection table is a map of all the subsystems in the loop and how they connect to each other.





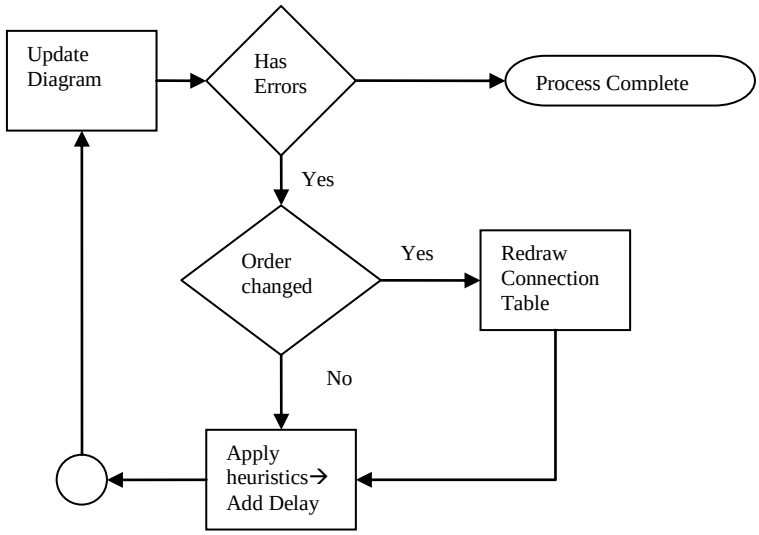
	A	C	B
A	NA	Output Input	Output
C	Input Output	NA	Input
B	Input	Output	NA

- The order the subsystem appear is in the column. The row header is based on their appearance in the error message
- Input: The row subsystem receives an Input from the column subsystem
- Output: The row subsystem has an Output that goes to the column subsystem

- NA: The column / row are the same subsystem (e.g. A/A)
- None: There are no connections between the row and column subsystem
- *: There is a connection however it is already broken by a unit delay; e.g. Input* indicates that the row subsystem is connected to the column subsystem
- If included the cells above the NA diagonal are a mirror of the cells below the NA diagonal.
- Length (*optional column*): A metric used to determine the distance between blocks in the loop

Heuristics for Using the Connection Table

The loops are broken by adding in unit delay blocks to the model. The following guidelines illustrate how to priorities the addition of unit delays. After each step of inserting unit delay the user should perform an update diagram. If the order of the subsystems in the error message changes the connection table should be re-drawn.



For larger models with longer update times the several sets of unit delays can be added between update diagrams. However as the size of the error loop decreases it is important to run the update diagram frequently to avoid the inclusion of unnecessary unit delays.

The basic premises of the heuristic are

- Break the shortest loops first
    - Break the loop between the subsystems where the direction changed.
        - Add to the subsystem with the most "resolved" signals. A signal is resolved if
            - It is a root level inport or constant block
            - It comes from a unit delay
            - It comes from a subsystem with no data dependencies.
        - The subsystem closer to the start of the loop

- For multiple co-existing loops once the zero length pairs are broken fixes the dependencies prior to the break first, working from the row before the break to the top of the table.

The connection table shows where loops exist. For a given row when the cell contents changes from input to output there is a loop between the subsystems. In the example table bellow a loop exists between

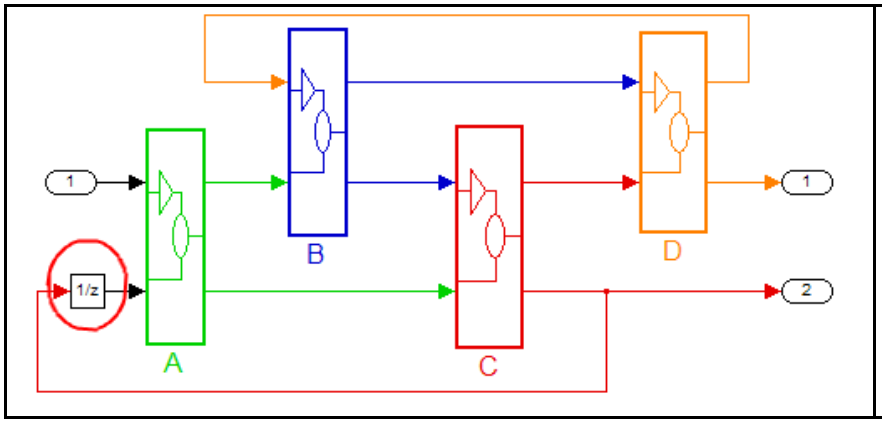
- F and G: The change from Input to Output (F/H → F/G)
- E and H: The change from Output to Input (E/F → E/H)
  - Loop would be broken between F & H
- G and H: The change from Output to Input in the same cell (H/G)
  - Loop would be broken between H & G
- G and E: The change from Input to Output (G/F → G/E)
  - Loop would be broken between F & E

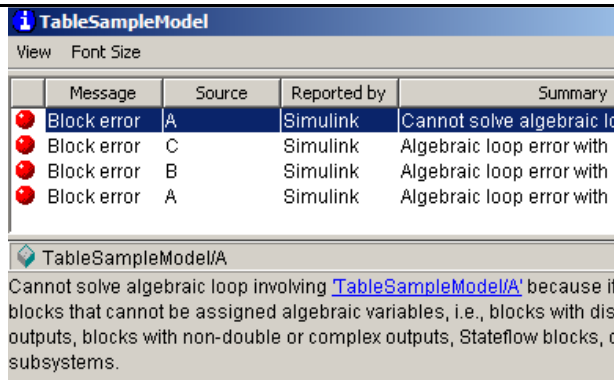
	F	E	H	G	Length
F	NA	Input	Input	Output	1
E	Output	NA	Input	Input	2
H	Output	Output	N A	Output Input	0
G	Input	Output	Input Output	NA	0,1

The length is determined by counting the number of columns between the start of one direction and the next.

Applying the Heuristics: A Direct Backwards Loop

Example 1: This example uses the model "Loop_E1.mdl"
Using the model and table from the "Building a connection table" section we apply the heuristics.



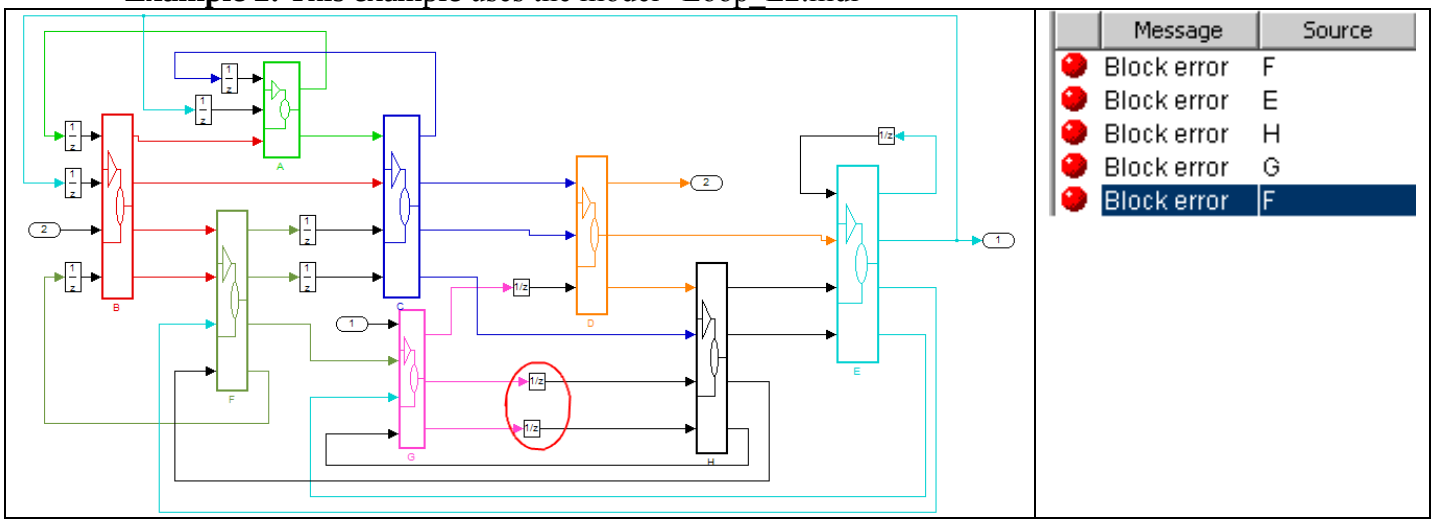


	A	C	B

A	NA	Output Input	Output
C	Input Output	NA	Input
B	Input	Output	NA

- The shortest loop is between A / C
  - The unit delay is added to the input of subsystem A.
  - A is selected because it has a root level input.
- Update Diagram: The loops are resolved.

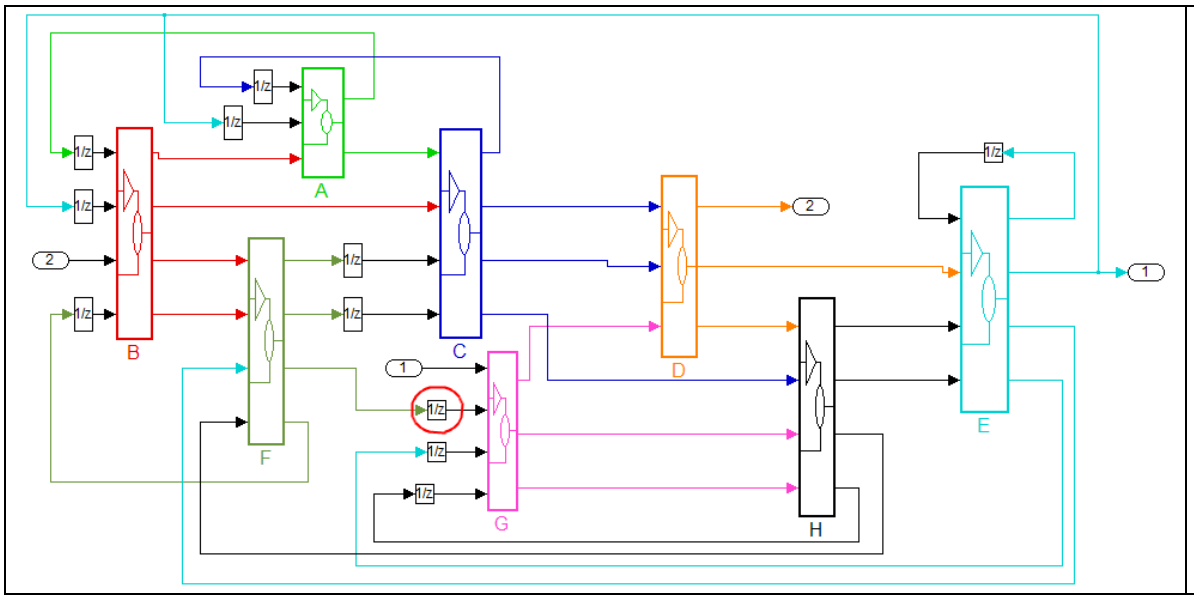
Example 2: This example uses the model "Loop_E2.mdl"



	F	E	H	G	Length
F	NA	Input	Input	Output	1
E	Output	NA	Input	Input	1
H	Output	Output	N A	Output Input	0
G	Input	Output	Input Output	NA	0

- The shortest loop exists between H & G
  - The unit delays are added to the inputs of subsystem H.
  - H is selected because it has the most resolved signals. The subsystem C and D are fully resolved, where as E and F (the inputs to G) are not.

Example 3: This example uses the model "Loop_E3.mdl"



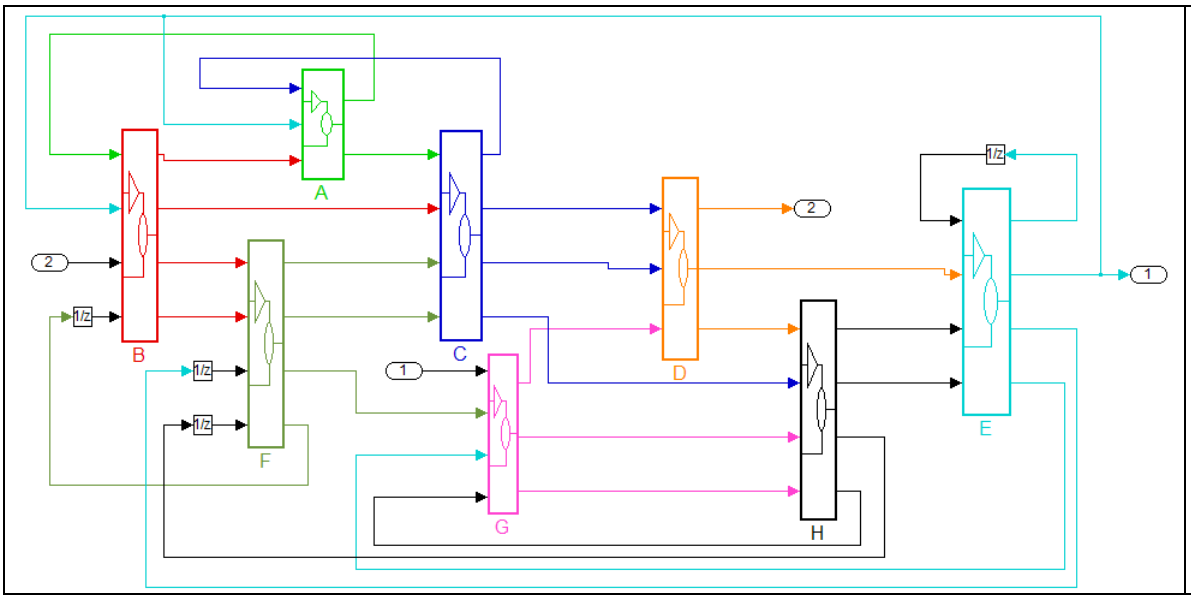
	H	D	G	F	E
H	NA	Input	Output* Input	Output	Output
D	Output	NA	Input	None	Output
G	Input* Output	Output	NA	Input	Input*
F	Input	None	Output	NA	Input
E	Input	Input	Output*	Output	NA

- The shortest loop is between F and G (F/H (input) → F/G (output))
  The H/G Output / Input pair is already broken by a unit delay
  - The unit delay is added to the inport of G.
- Update diagram: The loop is resolved

Applying the Heuristics: Multiple Coexisting Loops

In the first case the error message was easily interpreted because there was only one loop in the system. When there is more then one loop the error message maybe less clear. However the same methods can be used to determine useful information.

Example 4: This example uses the model Loop_E4.mdl



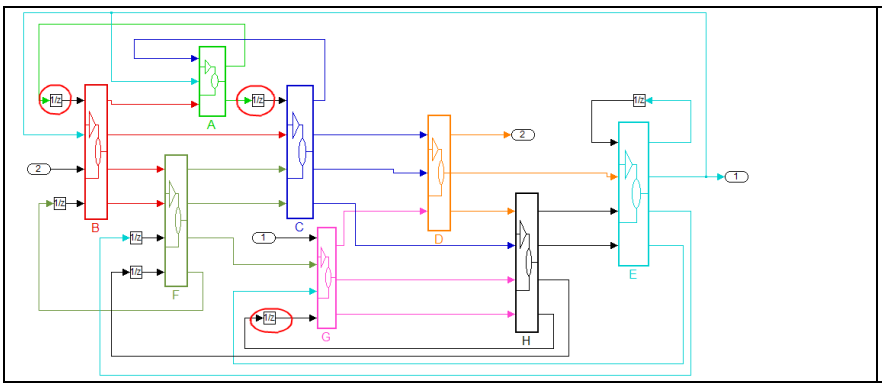
The update results in the following table. The gray shaded boxes represent subsystem pairs that are directly part of the error message loop. The table has four subsystems that have both inputs and output connections; A/C, B/A, G/H and F/B. The subsystem pair F/B can be ignore since the output connection is already broken by a unit delay.

	H	D	C	A	E	B	F	G
H	NA	Input	Output	None	Output	None	Output*	Output Input
D	Output	NA	Input	None	Output	None	None	Input
C	Output	Output	NA	Output Input	None	Input	Input	None
A	None	None	Input Output	NA	Input	Output Input	None	None
E	Input	Input	None	Output	NA	None	Output*	Output
B	None	None	Output	Input Output	Input	NA	Output* Input	None
F	Input*	None	Output	None	Input*	Input Output*	NA	Output
G	Input Output	Output	None	None	Input	None	Input	NA

- Shortest loop: G/H
  - Inserted break on inport to G.
- Update diagram: No change to error message
- Shortest loop: B/A
  - Inserted break into B since 2 out of three inputs already broken
- Update diagram: No change to error message
- Shortest loop: A/C

- Inserted break before Inport for C. Both A & C
  - Additionally, coupled with the previous step, results in the A row having all "Input" entries which removes it from the loop.

- Update diagram: Error message is updated
- Create new connection table



Message	Source
Block error	H
Block error	D
Block error	C
Block error	B
Block error	E
Block error	F
Block error	G
Block error	H

	H	D	C	B	E	F	G
H	NA	Input	Input	None	Output	Output*	Output* Input
D	Output	NA	Input	None	Output	None	Input
C	Output	Output	NA	Input	None	Input	None
B	None	None	Output	NA	Input	Output	None
E	Input	Output	None	Output	NA	Output*	Output
F	Input*	None	Output	Output* Input	Input*	NA	Output
G	Output Input*	Output	None	None	Input	Input	NA

The break in the connection table occurs at F/E. The error message shows a loop even though the loop is broken by a unit delay. Because of this we start by working on entries in rows H~E, prioritizing the bottom of the table.

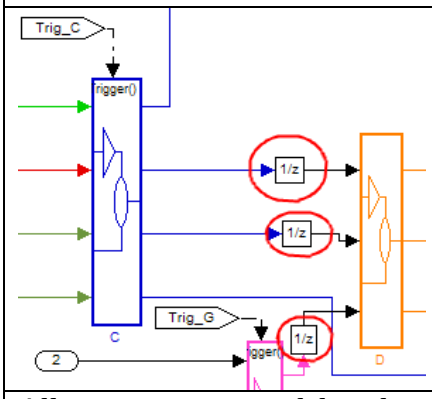
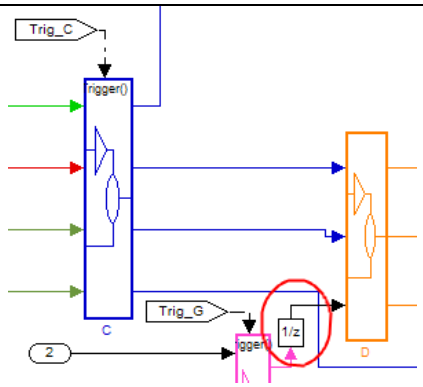
- Shortest loop: E & G (change from output at E/G to Input at E/H)
  - Inserted break on the input to G.
- Update diagram: No change to error message
- Shortest loop: B & E (change from output B/C to input B/E)
  - Inserted break on the input to B
- Update diagram: No error messages

Remove Unnecessary Unit Delays

A common problem when manually controlling the execution order is the introduction of unnecessary unit delays. The difficulty arises from determining which unit delays are required. The appendix section on data independent subsystems addresses how to remove unnecessary unit delays.

Mixed Function Call and Manual Control

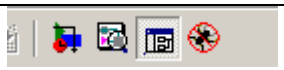
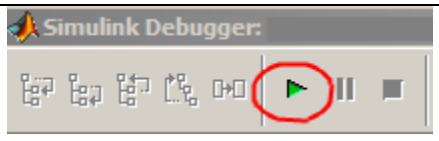
Using a combination of Function called and Manual control can be done, however it is not a recommended approach. Lets assume in our example model that A, C, E, G are controlled by function calls and B, D, F, and H are controlled by unit delays. Simulink requires that all function calls subsystems triggered by a common function call initiator execute without interruption. In our case that means that once A is triggered, C,E and G must run before any other subsystem runs. B, D, F and H could execute either before or after A, C, E and G.

Correct	**Incorrect**
	
All inputs into D are delayed. This allows subsystem D to run prior to the function call blocks.	The unit delay between G and D will cause D to run before G. However since there are no delays between C and D it should run after C.

Model Analysis

Verification of Execution Order for Manual Control Method

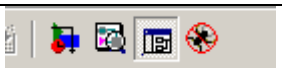
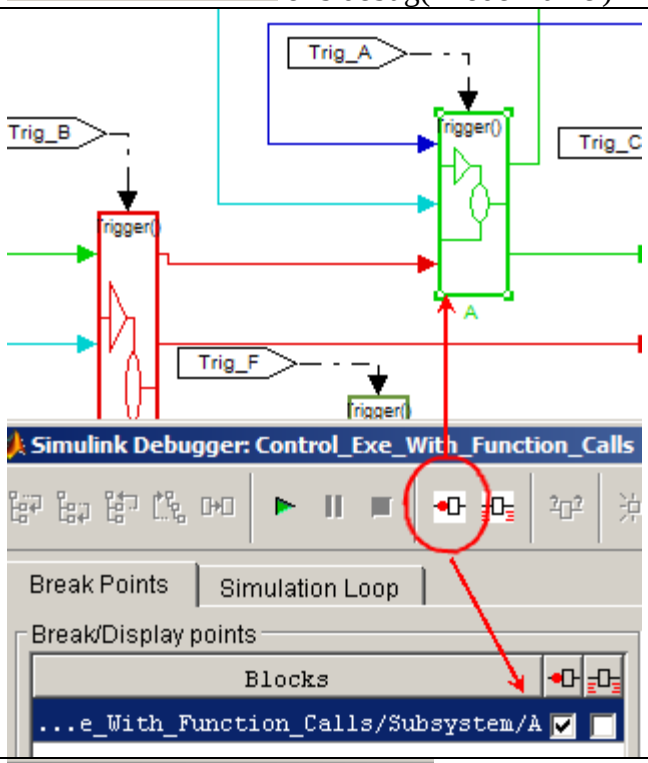
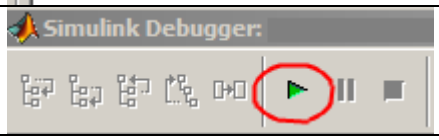
If the manual method for controlling execution has been used then the Simulink Debugger can be used to find the order of execution of the subsystems.

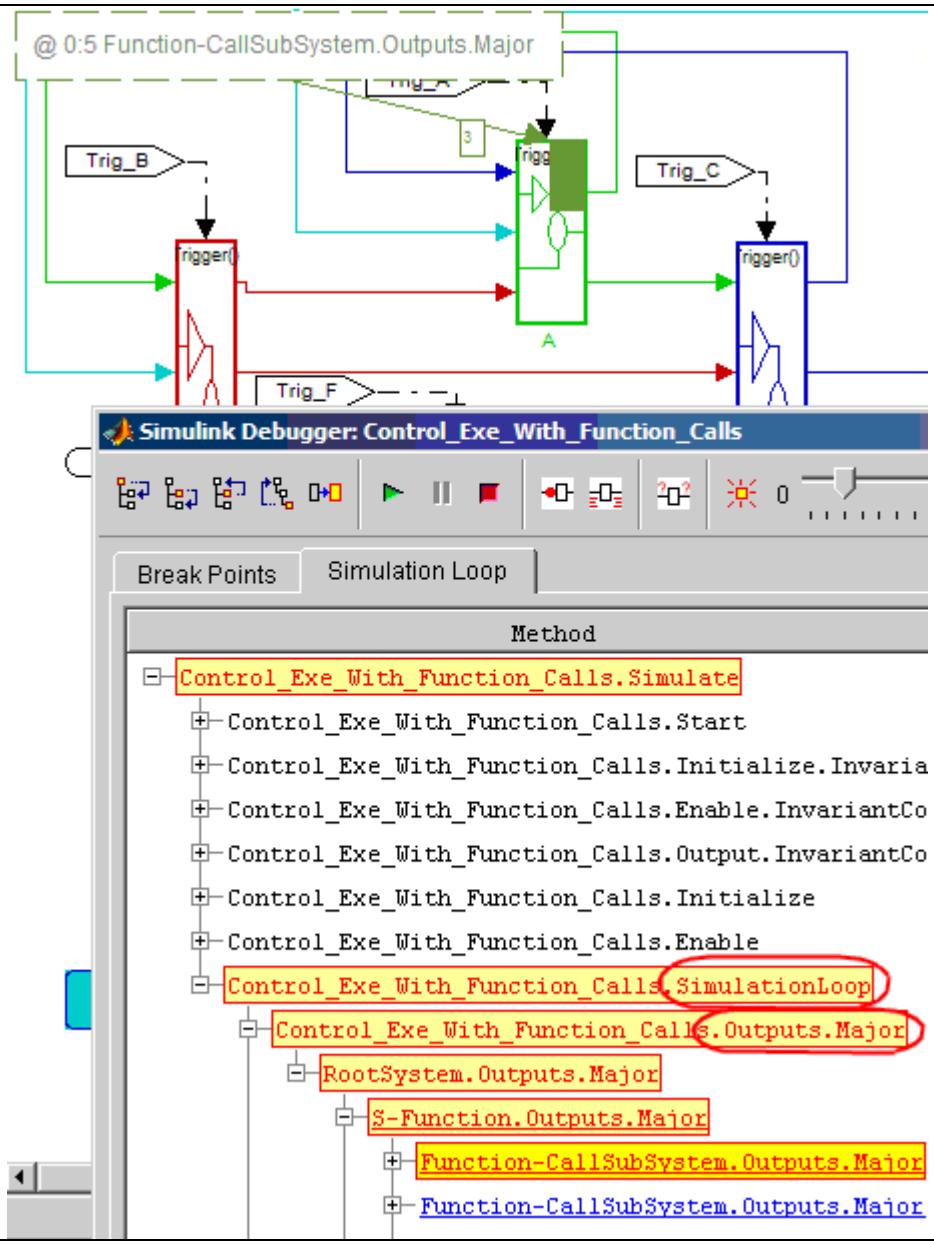
Launch the debugger	 or sldebug('ModelName')
Start the simulation from the debugger	
Open the Sorted List Tab	

View the sorted list **Note:** this list is filtered to just show the subsystems	`---- Sorted list for 'Control_Exe_Base_full_manual' [25 nonvirtual`
	`  0:4  'Control_Exe_Base_full_manual/A' (SubSystem)`
	`  0:7  'Control_Exe_Base_full_manual/B' (SubSystem)`
	`  0:10  'Control_Exe_Base_full_manual/C' (SubSystem)`
	`  0:12  'Control_Exe_Base_full_manual/D' (SubSystem)`
	`  0:15  'Control_Exe_Base_full_manual/E' (SubSystem)`
	`  0:19  'Control_Exe_Base_full_manual/F' (SubSystem)`
	`  0:21  'Control_Exe_Base_full_manual/G' (SubSystem)`
	`  0:24  'Control_Exe_Base_full_manual/H' (SubSystem)`

Verification of Execution Order for Function Call Driven Method

Unlink the unit delay case the Simulink debugger will not directly give the execution order of the subsystems for function call triggered subsystems. However by using break points the debugger can be used to determine the order of execution.

Launch the debugger	 or sldebug('ModelName')
For each atomic subsystem select the block and then add it to the break point list	
Start the simulation from the debugger	

Advance the simulation until the model is in the "SimulationLoop" "Outputs.Major" sub-group. Each step of the model will now advance to the next active subsystem. In the image A is the current active subsystem.	

Appendix

Understanding Execution Order

How Simulink Determines the Sorted Order

Simulink uses the following basic rules to sort the blocks: Each block must appear in the sorted order ahead any of the blocks whose direct-feed through ports (see About Direct-Feed through Ports) it drives. This rule ensures that the direct-feed through inputs to blocks will be valid when block methods that require current inputs are invoked. Blocks that do not have direct feed through inputs can appear anywhere in the sorted order as long as they precede any blocks whose direct-feed through inputs they drive. Putting all blocks that do not have direct-feed through ports at the head of the sorted order satisfies this rule. It thus allows Simulink to ignore these blocks during the sorting process. The result of applying these rules is a sorted order in which blocks without direct feed through ports appear at the head of the list in no particular order followed by blocks with direct-feed through ports in the order required to supply valid inputs to the blocks they drive. During the sorting process, Simulink checks for and flags the occurrence of algebraic loops, that is, signal loops in which a direct-feed through output of a block is connected directly or indirectly to the corresponding direct-feed through input of the block. Such loops seemingly create a deadlock condition, because the block needs the value of the direct-feed through input to compute its output. However, an algebraic loop can represent a set of simultaneous algebraic equations (hence the name) where the block's input and output are the unknowns. Further, these equations can have valid solutions at each time step. Accordingly, Simulink assumes that loops involving direct-feed through ports do, in fact, represent a solvable set of algebraic equations and attempts to solve them each time the block's output is required during a simulation. For more information, see Algebraic Loops.

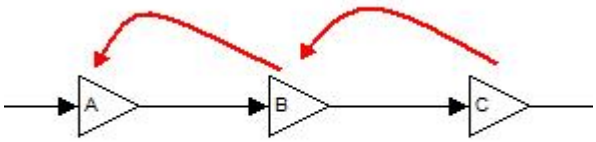
About Direct-Feed through Ports

In order to ensure that the sorted order reflects data dependencies among blocks, Simulink categorizes a block's input ports according to the dependency of the block's outputs on its inputs. An input port whose current value determines the current value of one of the block's outputs is called a direct-feed through port. Examples of blocks that have direct-feed through ports include the Gain, Product, and Sum blocks. Examples of blocks that have non-direct-feed through inputs include the Integrator block (its output is a function purely of its state), the Constant block (it does not have an input), and the Memory block (its output is dependent on its input in the previous time step).

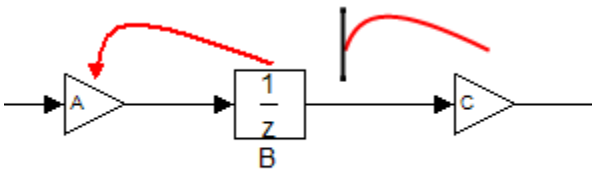
Data Dependency

Fundamentally Simulink uses data dependency to determine the order in which calculations are performed. If block **B** uses the output of block **A** then block **B** is said to

be dependent on block **A**. Dependency propagates through blocks, so if **C** uses **B**'s output then **C** depends on **A**.

	B(n) = f(A(n)) C(n)= f(B(n)) C(n) = f(f(A(n))

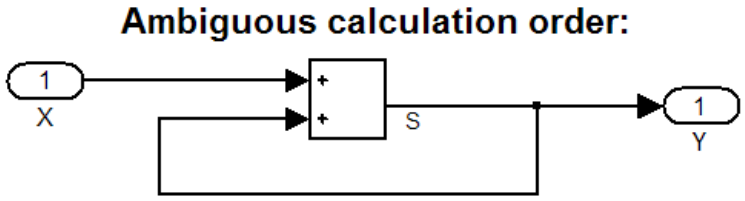
Blocks that maintain state information, such as integrators and unit delays break data dependency for the blocks following them.

	B(n) = f(A(n)) C(n) = f(B(n-1))

The same principal can be extended to Atomic Subsystems.

Basic Algebraic Loops

Algebraic loops are the result of Simulink block connections with ambiguous data dependency. Ambiguity arises when the output of a block is used, directly or through a chain of blocks, as an input to itself.



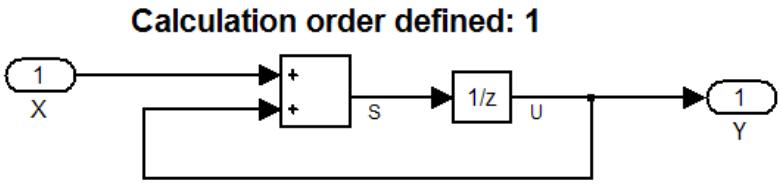
Note: The subscript letter "i" indicates the iteration count

$Y_i = S_i$

$S_i = X_i + S_i$

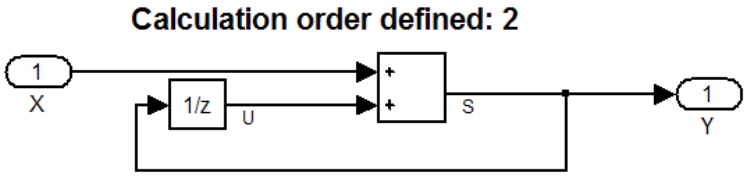
Subtracting S_i from both sides results in

$X_i = 0$



$Y_i = U_i$
$U_i = S_{i-1}$
$S_i = X_i + U_i$
By substitution
$Y_i = X_{i-1} + Y_{i-1}$

The resulting equation is not ambiguous; however the output result is based on a delayed value of the input X.

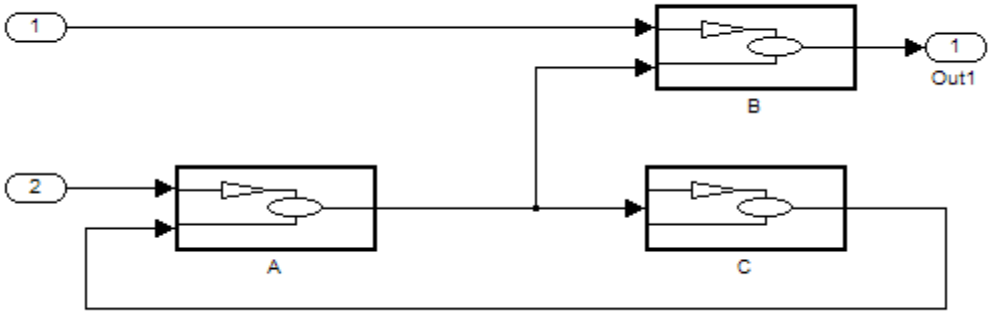


$Y_i = S_i$
$S_i = X_i + U_i$
$U_i = S_{i-1}$
By substitution:
$Y_i = X_i + S_{i-1} = X_i + Y_{i-1}$

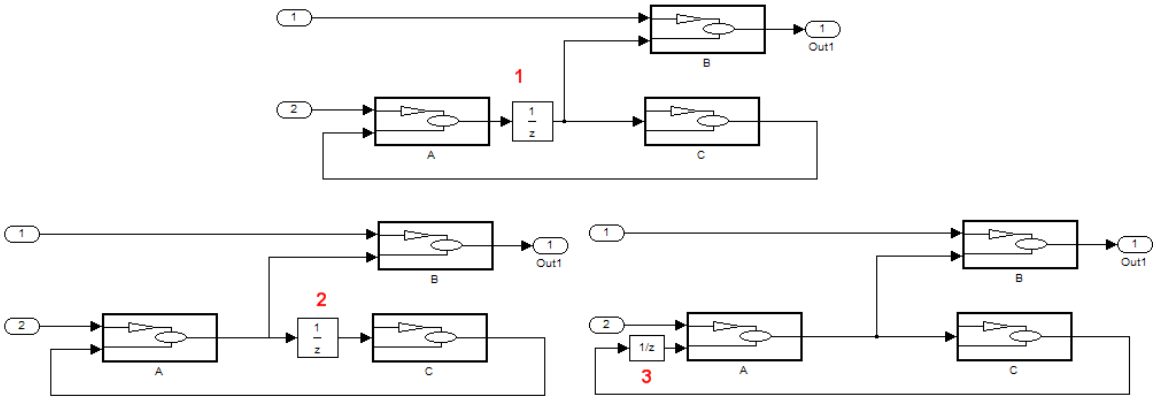
This configuration of the block diagram results in the desired calculation; the output value is based on the current value of X and the last value of Y.

The Affect of Breaking Data Dependencies on System Behavior

As the previous section showed unit delay blocks can be used to resolve data dependencies. At the system root level the data dependencies will exists between the atomic subsystems.



In the figure above there is a data dependency loop between subsystems A and C; there is a data dependency between A and B but there is not a loop. There are three different ways in which the loop can be broken.



The placement of the unit delay affects both the order of execution and the TIME SLICE of the data consumed by the subsystem.

	Method 1	**Method 2**	**Method 3**
Execution Order	1. B / C 2. A	1. C 2. A 3. B	1. A 2. B / C
A Inputs	In 1 current In 2 current	In 1 current In 2 current	In 1 current In 2 last pass
B Inputs	In 1 current In 2 last pass	In 1 current In 2 current	In 1 current In 2 current
C Inputs	In 1 last pass	In 1 last pass	In 1 current

For both methods 1 and 3 subsystems B & C are show as executing at the same time. This is because there is no data dependency between them. They can be executed in any order without changing the simulation behavior.

Examining this simple model yields two rules basic guidelines.

1. Break dependency loops at the sink
   Breaking dependencies at the source (for example Method 1) introduces delays in the signal that may not be required (A to B)
2. Place unit delays such that the time frame of the inputs are consistent.

Because of these guidelines the recommendation is to use method 2.

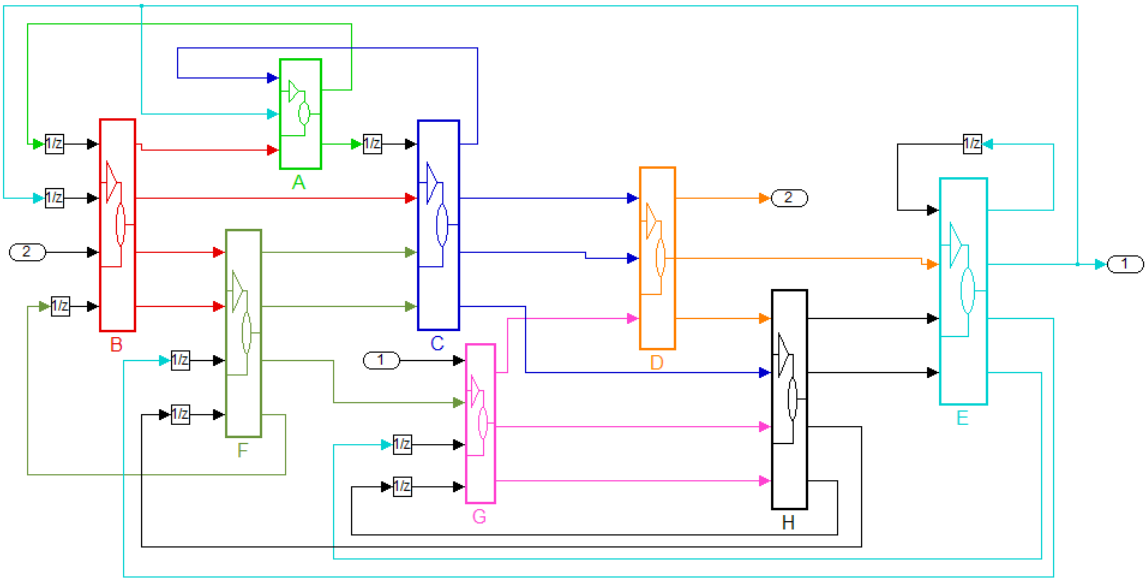
Data Independent Subsystems

A subsystem with no input data dependences or output data dependences is called data independent. A subsystem is input data independent if all of the inputs are either root level inports or unit delayed signals. These subsystems run at the start of the time step. They should not have any delays on their output signals. Subsystems receiving signals from an input data independent subsystem will run after the source.

If all the input signals to a subsystem are from a combination of input data independent subsystems, root level inputs and unit delayed signals then the subsystem is called inherited data independent. These subsystems act like data independent subsystems with respect to the down stream subsystems. However they may have unit delays on their outputs.

A subsystem is output data independent if all the outputs are either delayed or root level inports. These subsystems will run near the end time step. These subsystems should not have any unit delays on their inports.

Understanding which subsystems are data independent facilitates the removal of necessary unit delays. We will start off with an example model that does not have any extra unit delays to illustrate this concept.



In the model above the subsystem B is input data independent; 3 out of its four inputs are delayed and the fourth is a root level input. Subsystem E is not data independent because

of the connection to subsystem A. Examining the remaining subsystems, proceeding left to right across the diagram we observe the following

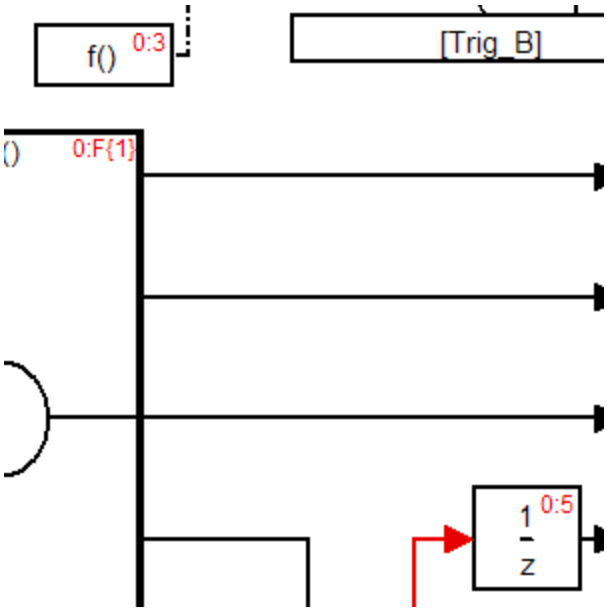
- F is dependent on B, E and H.
    - The signals from E & H are directly broken with unit delays.
    - B is a data independent subsystem
      Therefore
    - F is an inherited data independent.
    - F will run after B, before E & H
- A is dependent on C, E and B.
    - For now A is uncertain.
- C is dependent on A, B and F
    - The signal from A is broken with a unit delay
    - Signals from F and B are inherited and data independent signals
      Therefore
    - C is an inherited data independent subsystem
    - C will run after B and F, and before A.
- G is dependent on F, E and H
    - The signals from E and H are broken by unit delays
    - F is an inherited data independent subsystem
      Therefore
    - G is an inherited data independent subsystem
    - G will run after F, before H & E.
    - The execution order of G & C are independent of each other.
- By inspection D, H and E are inherited data independent subsystems.
- Once E is resolved A can be stated to be inherited data independent.

Guidelines for removing unnecessary unit delays

- Subsystems should not have both input and output unit delays
- The majority of unit delays should be near the start of the signal flow (e.g. the left hand side of the diagram)
    - Unit delays near the right hand side of the diagram should be to resolve local loops

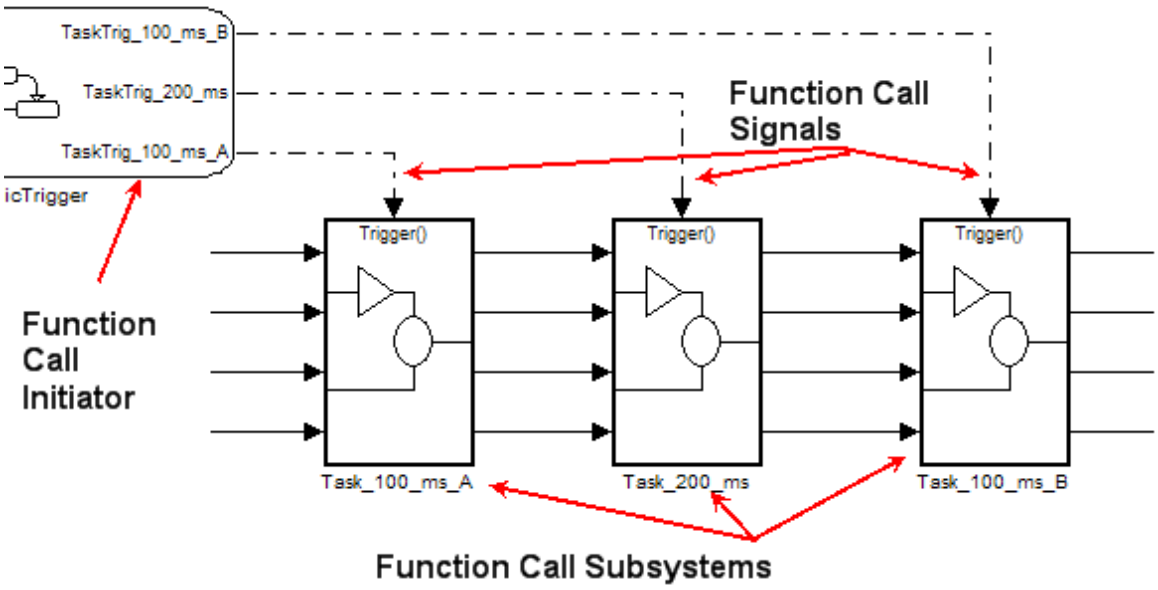
Simulink Sorted Order

The figure below shows part of a Simulink model with the block sorted order turned on. (Simulink >> Format >> Block Displays >> Sorted Order) It is important to note that the number shown for the subsystem "0:F{1}" is not the execution order of the system. The F{1} denotes it as the 1st subsystem. Blocks contained in that subsystem take the label 1:N where N is there location in the order of execution.

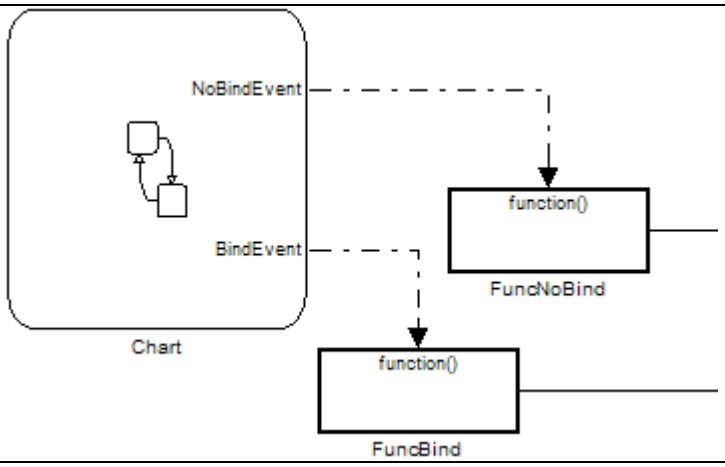
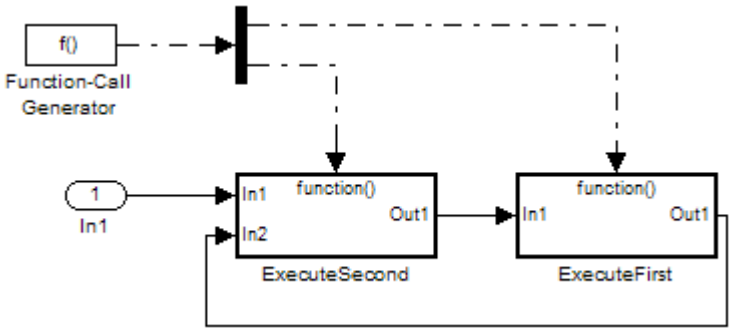
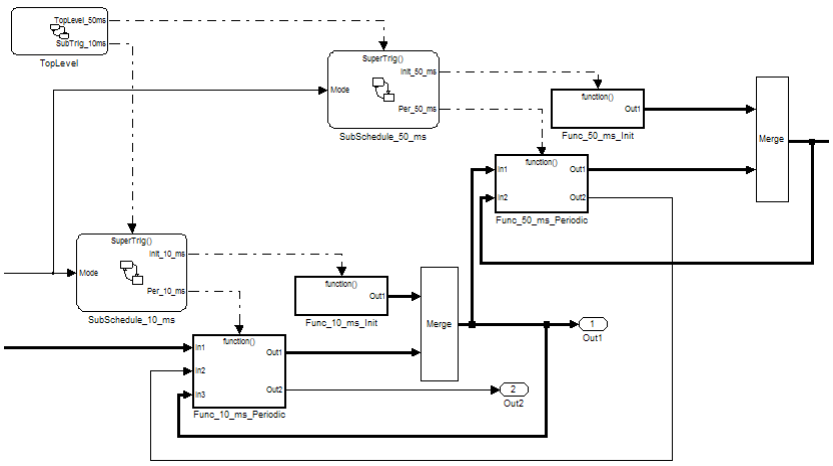


Function Call Initiators

Function call subsystems are triggered by function call signals. Function call signals are generated by function call initiators. Stateflow and Function Call Generator blocks are the most common methods for creating a function call signal.



Common function call initiators are either direct or indirect. In direct common function call initiators all the subsystems are directly triggered by the function call source. Indirect common function call initiators use a root function call source to trigger sub-function call sources.

	Direct Common Function Call Initiator Single Stateflow diagram
	Direct Common Function Call Initiator
	Indirect Common Function Call Initiator • The Stateflow diagram "TopLevel" triggers the two sub schedulers.

For the Function Call Driven approach to work the model must use one common function call initiator for each rate / offset.

Atomic Subsystems and Execution Order

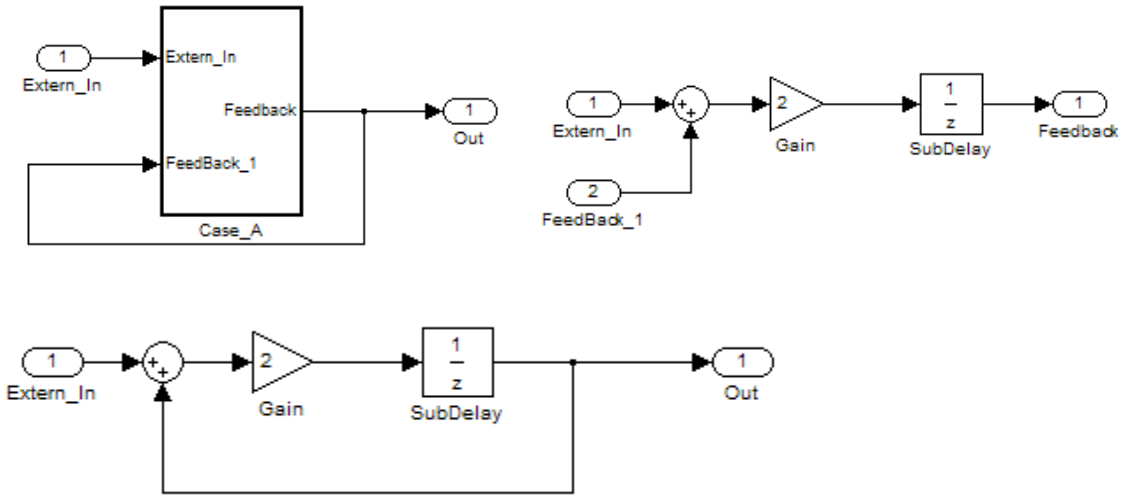
Subsystems in Simulink are either Virtual or Atomic. Virtual subsystems are a method of visually organizing the model and have no affect on either the simulation behavior the generated code. Designating a subsystem as atomic forces Simulink to treat the subsystem as a unit when determining the execution order of block methods. For example, when it needs to compute the output of the subsystem, Simulink invokes the

output methods of all the blocks in the subsystem before invoking the output methods of other blocks at the same level as the subsystem block.

If this option is not selected, Simulink treats all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem when determining block method execution order. This can cause execution of methods of blocks in the subsystem to be interleaved with execution of methods of blocks outside the subsystem

Designating the subsystem as atomic provides control over how the code is generated; inline, function or reusable function. Additionally the subsystems data can be placed into unique data structures if the code is either function or reusable function.

Changing a subsystem from virtual to atomic *affects how Simulink determines the data dependency of the system.* Because of this the recommendation is to only use atomic subsystems the partitioning control is required. The next section goes into detail on how atomic subsystems affect the model's behavior.



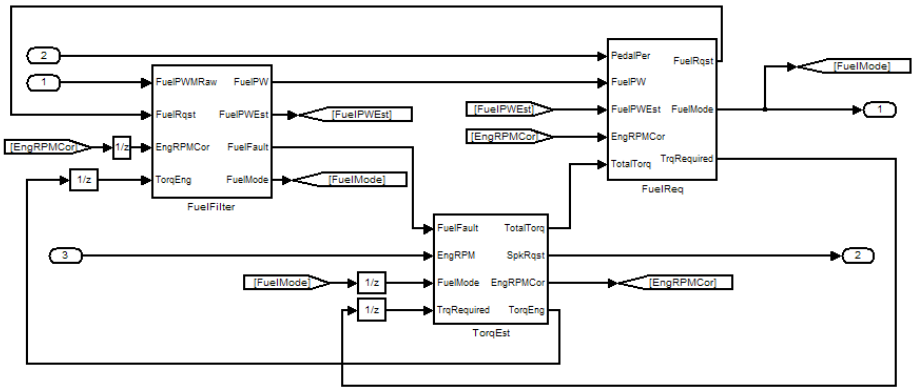
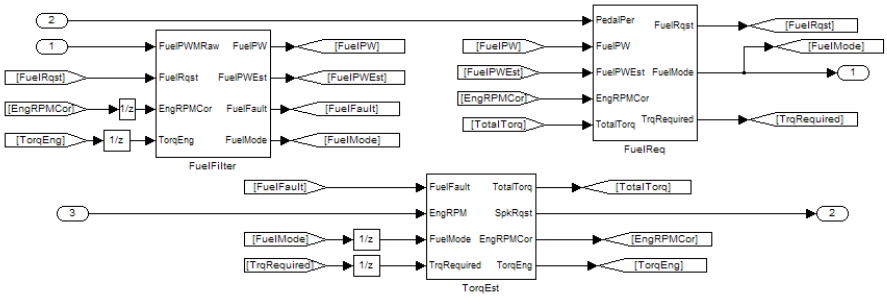
In the example above, if the subsystem Case_A is a virtual subsystem then the full model reduces to a system without data dependencies. However if the subsystem Case_A is an atomic subsystem then Simulink treats the system as having a data dependency. This sort of data dependency can be resolved by using the subsystem configuration option "Minimize algebraic loops." This method will only work if the data contains a block that has state information. More information on how this option works can be found in the Simulink documentation.

Referenced MAAB Rules

The following MAAB style guide rules where referenced in this document. They are included here for reference only.

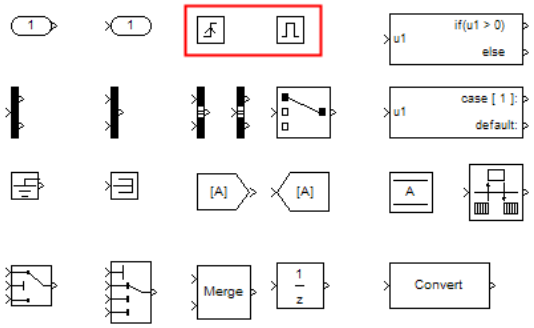
jc_0171: Maintaining signal flow when using Goto and From blocks

ID: Title	**jc_0171: Maintaining signal flow when using Goto and From blocks**
Priority	strongly recommended
Scope	MAAB
MATLAB Version	All
Prerequisites	

Description	• Visual depiction of signal flow must be maintained between subsystems
	• Use of Goto and From blocks is allowed provided that
	  ◦ At least one signal line is used between connected subsystems
	  ◦ If the subsystems are connected both in a feedforward and feedback loop then at least one signal line for each direction must be connected
	Correct
	 
	
	Incorrect
	 

Rationale	☑ Readability ☑ Workflow ☐ Simulation	☑ Verification and Validation ☐ Code Generation
Last Change	V2.0	

db_0143: Similar block types on the model levels

ID: Title	**db_0143: Similar block types on the model levels**
Priority	strongly recommended
Scope	MAAB
MATLAB Version	All
Prerequisites	
Description	Every level of a model must be designed with building blocks of the same type. (i.e. only subsystems or only basic blocks). **Blocks which can be placed on every model level:** Inport Outport Enable (not on highest model level) Trigger (not on highest model level) Mux Demux Bus Selector Bus Creator Selector Ground Terminator From Goto Switch Multiport Switch Merge Unit Delay Rate Transition Type Conversion Data Store Memory If block Case block Note: Trigger and Enable blocks can not be placed at the root level. 
Rationale	☑ Readability ☑ Verification and Validation	

	☑ Workflow ☐ Code Generation ☐ Simulation
Last Change	V2.0