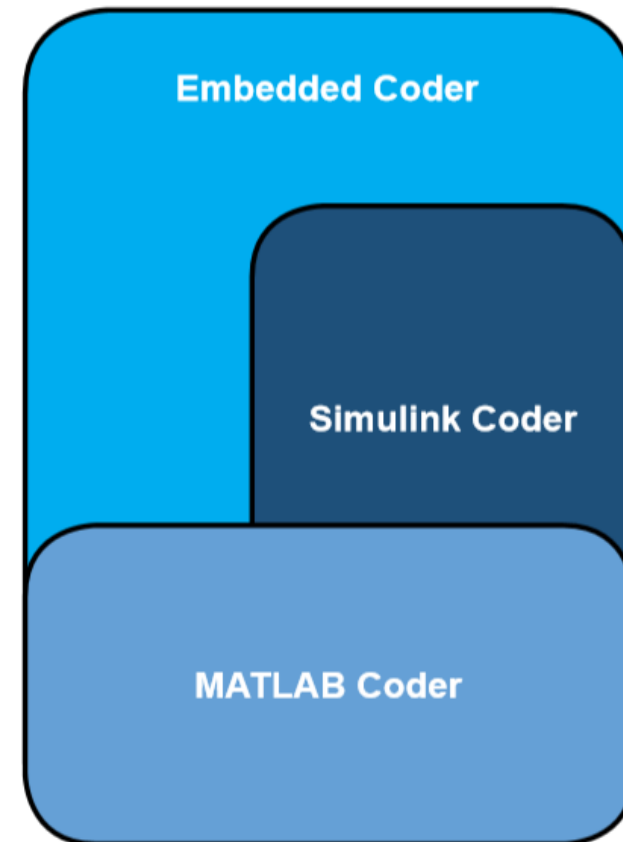
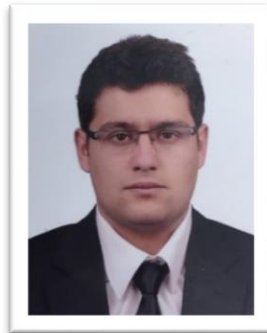


# Embedded Coder for Production Code Generation

**Erkam ÇANKAYA**  
Kıdemli MATLAB Uygulama Mühendisi



# Agenda

## Generating Embedded Code

- System specification
- Generating ERT code
- Code modules
- Logging intermediate signals
- Data structures in generated code
- Embedded Coder build process

## Optimizing Generated Code

- Optimization considerations
- Removing unnecessary code
- Removing unnecessary data support
- MAT-File Logging
- Inlined Parameters
- Code generation objectives

## Bus Objects and Model Referencing

- Bus signals and model referencing
- Controlling data type of bus signals
- Controlling storage class of bus signals

## Improving Code Efficiency and Compliance

- The Model Advisor
- Hardware implementation parameters
- Compliance with standards and guidelines

# Generating ERT code

>> iir

System equation in time domain:

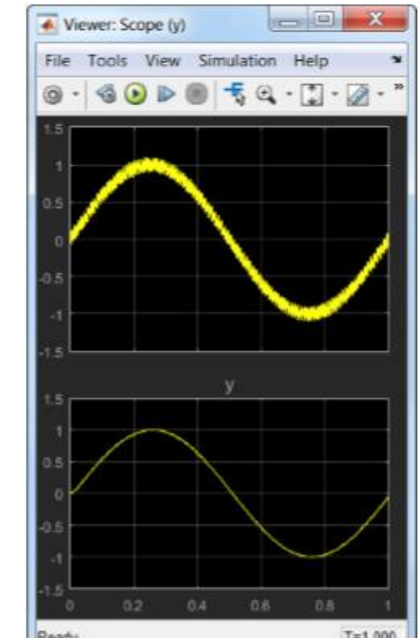
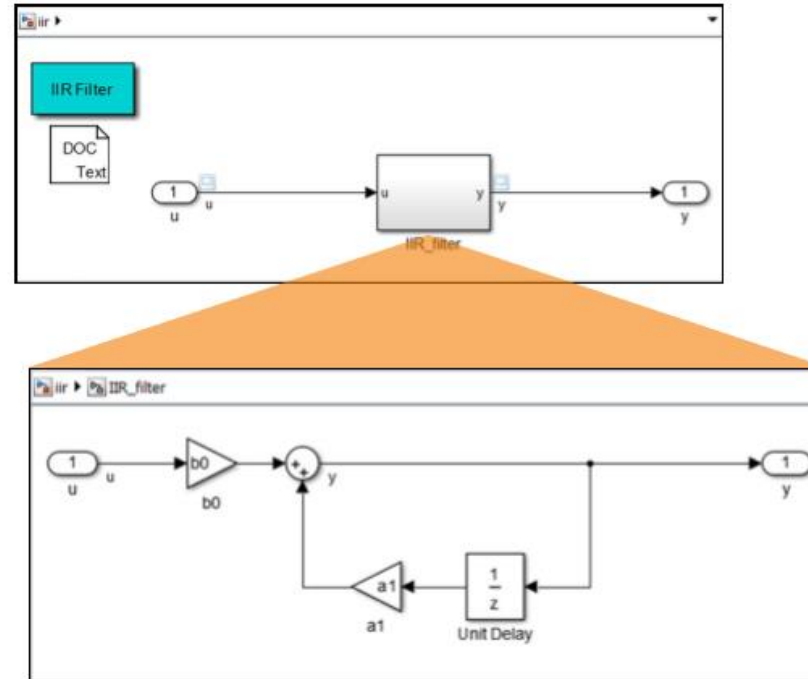
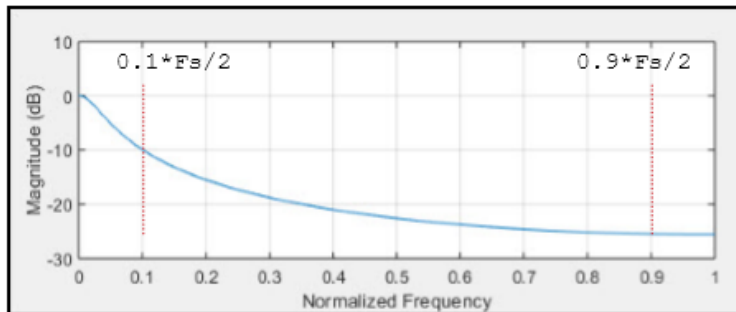
$$y[k] = b_0 \cdot x[k] + a_1 \cdot y[k - 1]$$

System equation in Z-domain:

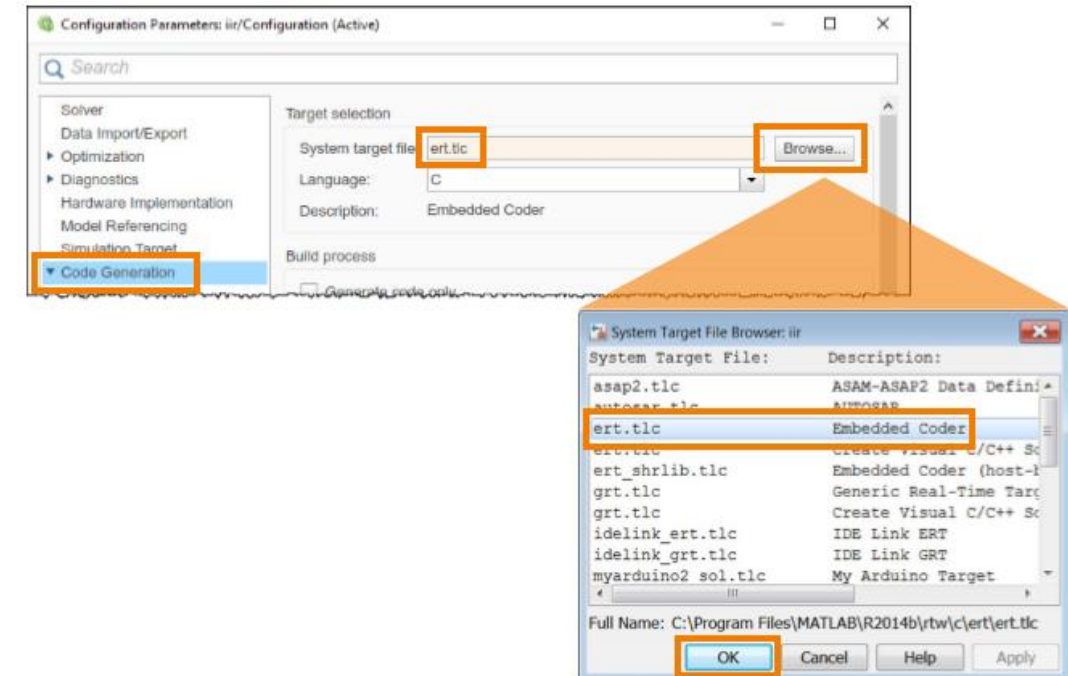
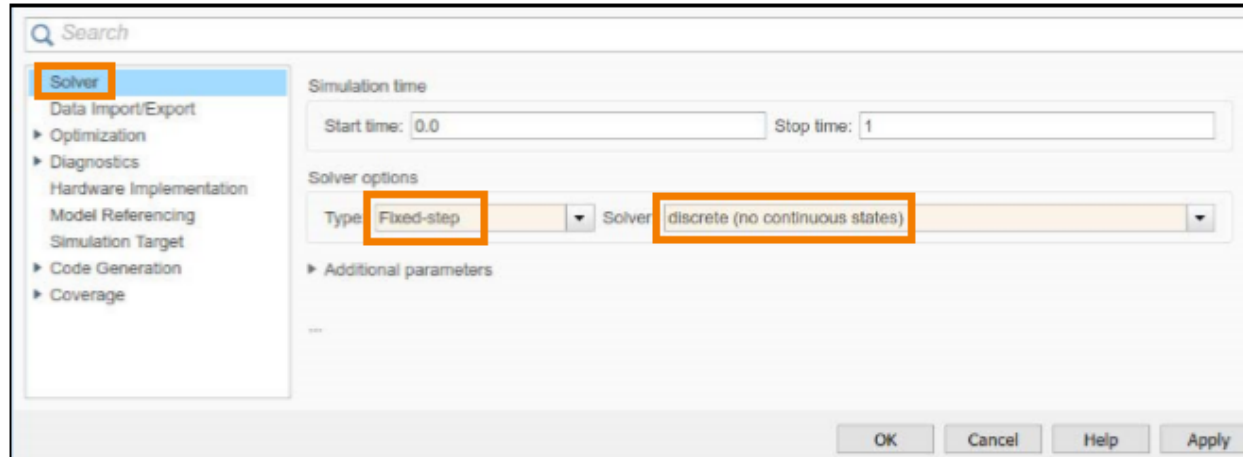
$$H(z) = \frac{b_0}{1 - a_1 z^{-1}}$$

Parameter-Set for filter gains:

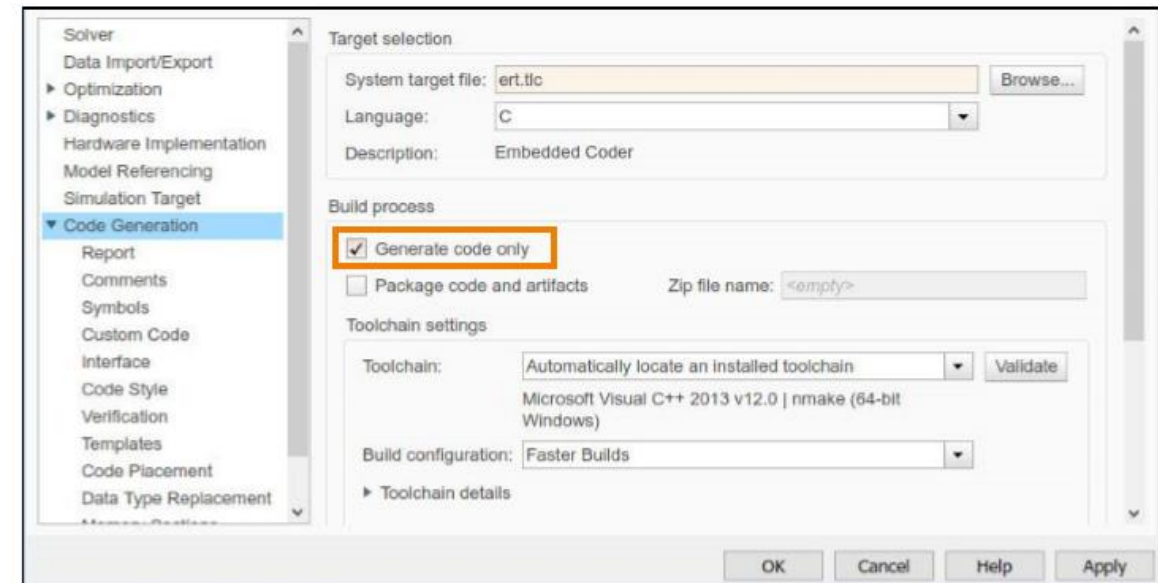
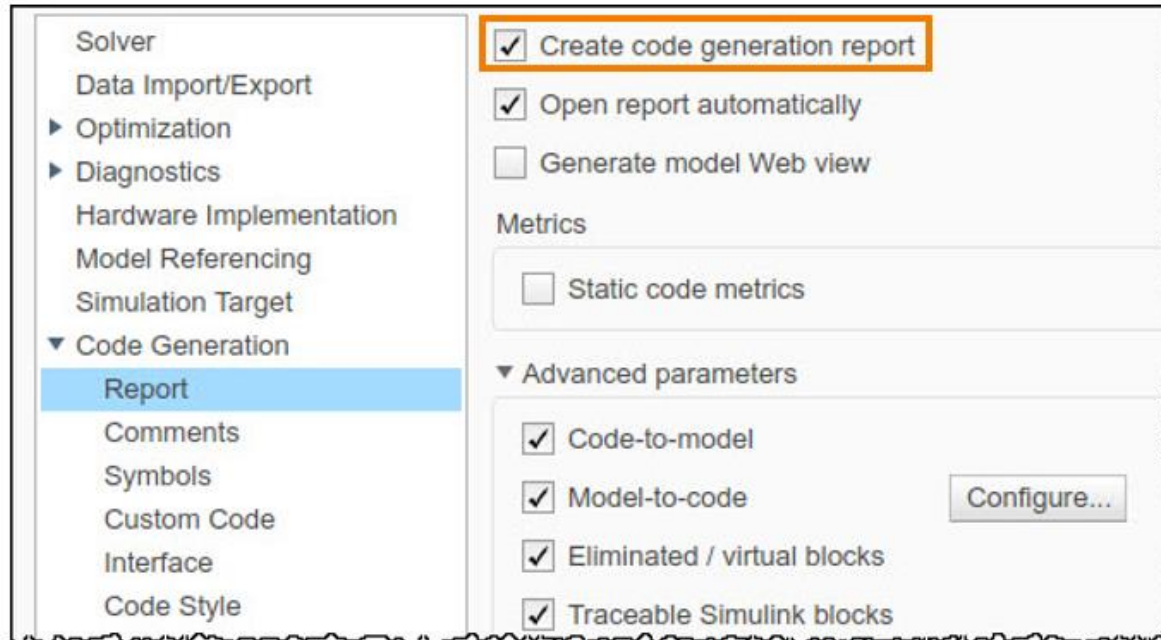
$$b_0 = 0.1 \quad a_1 = 0.9$$



# Generating ERT code

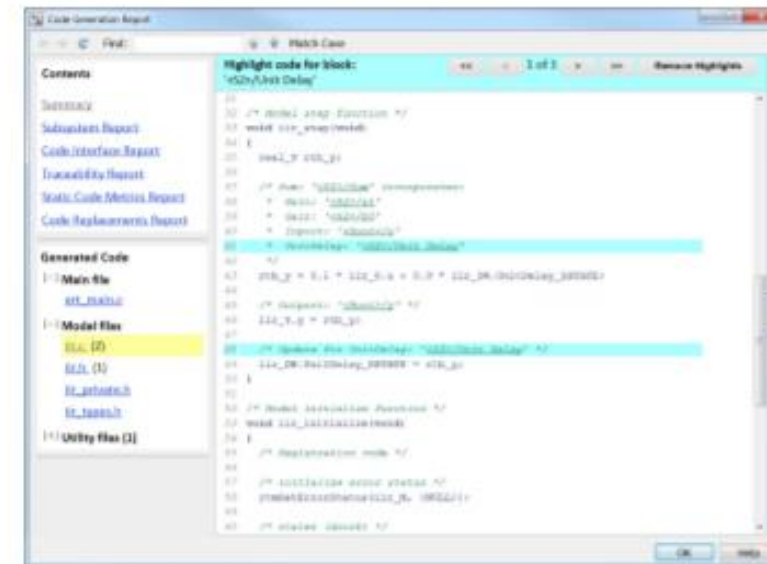
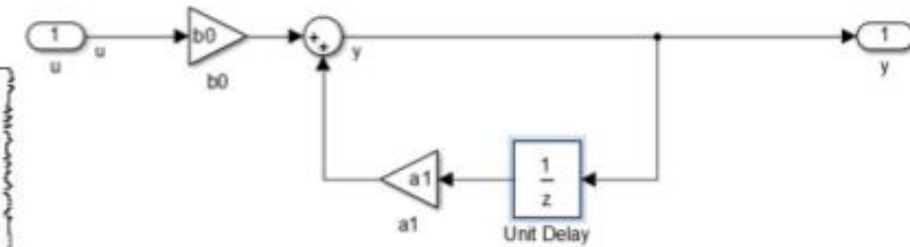
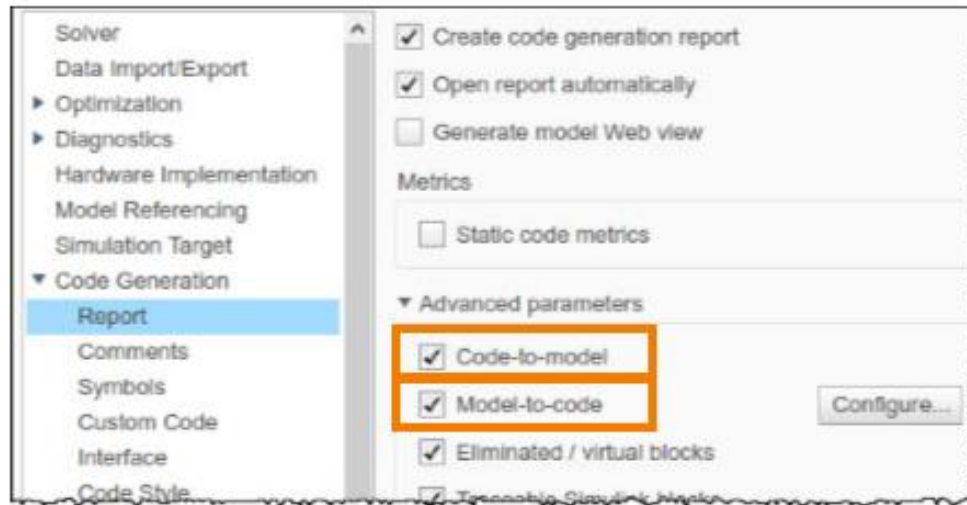


# Generating ERT code

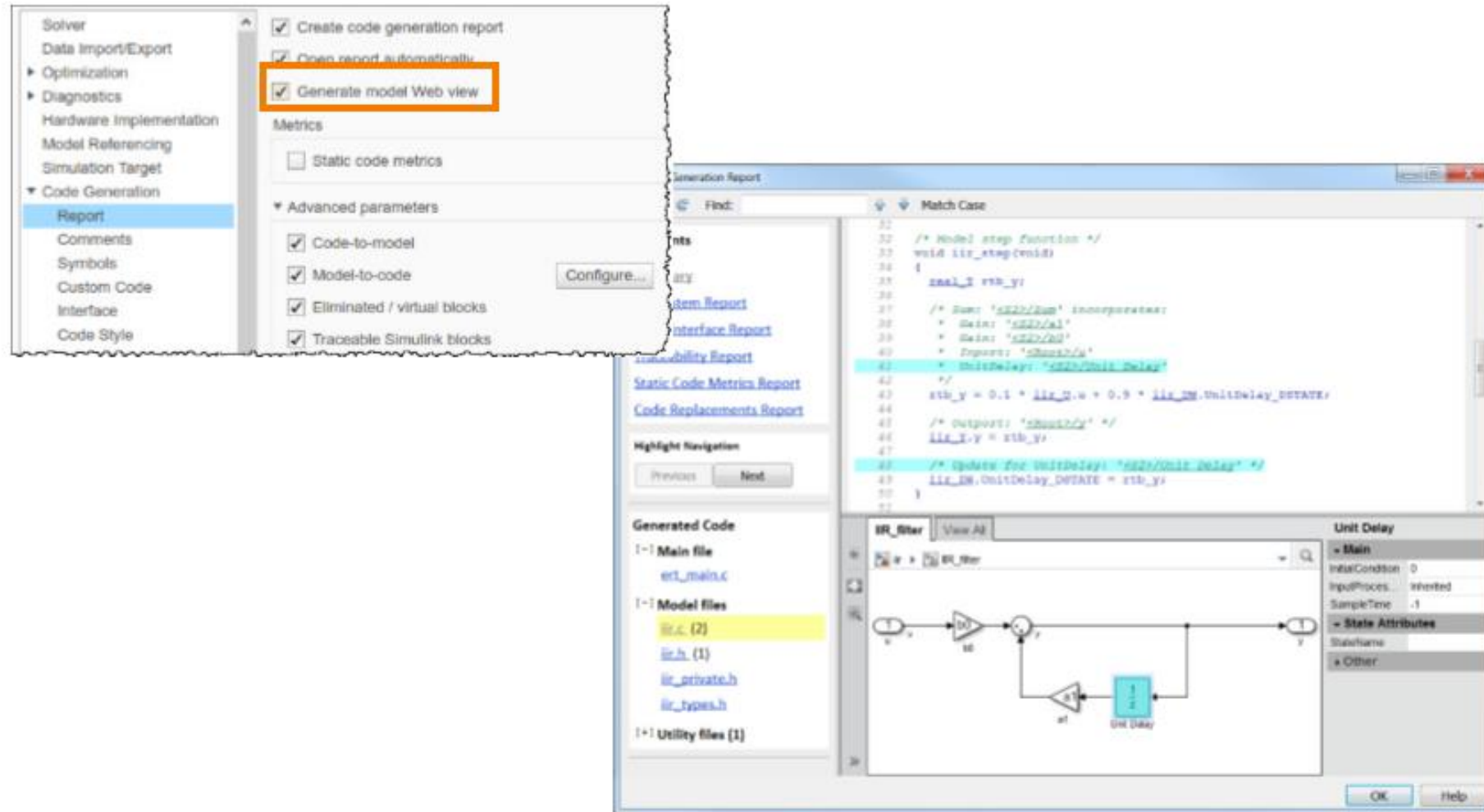


# Generating ERT code

## Bidirectional Traceability



# Generating ERT code



The screenshot shows the MATLAB/Simulink Code Generation interface. On the left, the 'Code Generation' pane is open, and the 'Generate model Web view' option is checked and highlighted with an orange box. Below this, the 'Advanced parameters' section shows several options checked: 'Code-to-model', 'Model-to-code', 'Eliminated / virtual blocks', and 'Traceable Simulink blocks'. The 'Generated Code' pane on the left lists the files generated for the 'IR\_Riser' model, including 'ert\_main.c', 'ir.c (2)', 'ir.h (1)', 'ir\_private.h', 'ir\_types.h', and 'Utility files (1)'. The main window displays the 'Generation Report' for the 'IR\_Riser' model, showing the C code generated for the 'Unit Delay' block. The code includes a function 'lir\_step' that calculates the output of the unit delay block based on the input and the current state of the delay.

```
>> license('inuse')
matlab
matlab_coder
matlab_report_gen
real-time_workshop
rtw_embedded_coder
simulink
simulink_report_gen
```



# Code modules

*model.c*: Contains entry points for all code implementing the model algorithm

*model.h*: Declares model data structures and a public interface to the model entry points and data structures. Also provides an interface to the real-time model data structure (*model\_M*) with accessor macros. If you are interfacing your manually-written code to generated code for one or more models, you should include *model.h* for each model to which you want to interface.

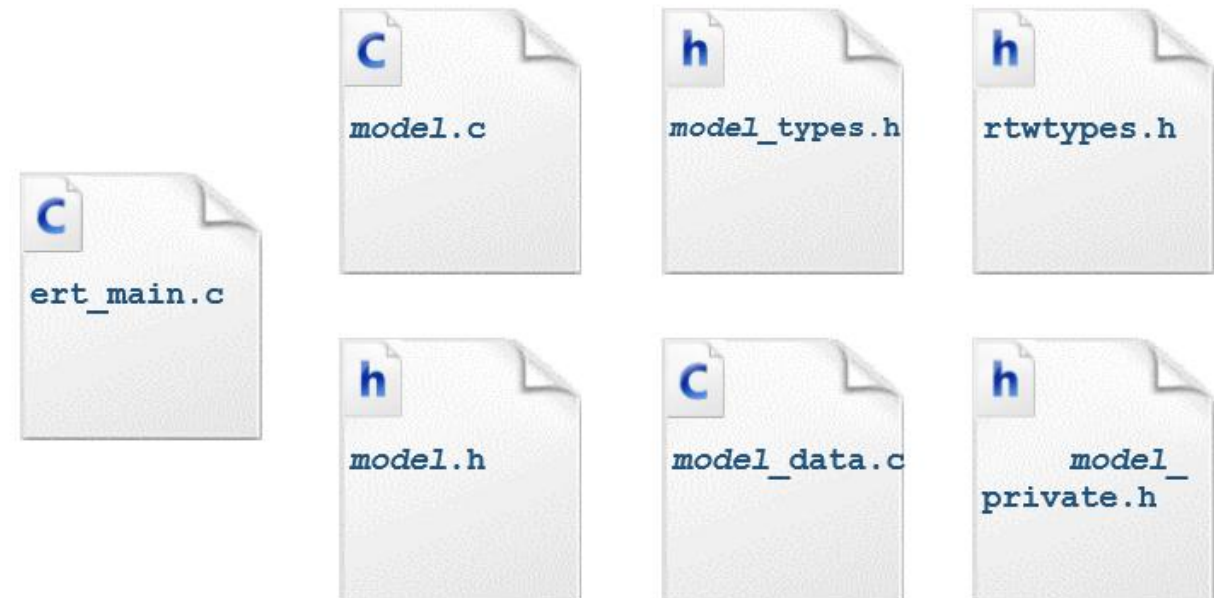
*model\_private.h*: Contains local macros and local data that are required by the model and subsystems. This file is included by the generated source files in the model.

*model\_types.h*: Provides forward declarations for the real-time model data structure and the parameters data structure. These may be needed by function declarations of reusable functions.

*rtwtypes.h*: Defines data types, structures and macros required by Embedded Coder. Most other generated code modules require these definitions.

*ert\_main.c*: This file is an example main program that Embedded Coder generates by default.

*model\_data.c*: This file is conditionally generated. It contains the declarations for the parameters data structure and the constant block I/O data structure. If these data structures are not used in the model, *model\_data.c* is not generated.

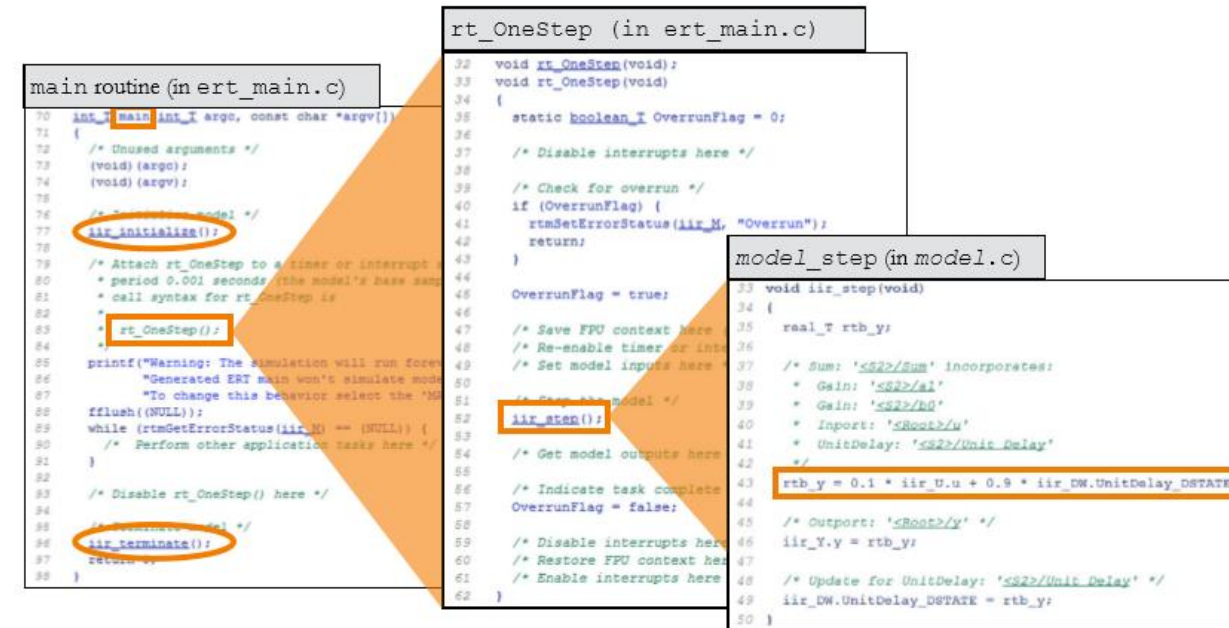




# Code modules

## The main Routine

- *model\_initialize*: Called at the beginning of the execution, implemented in *model.c*, and contains initialization code for:
  - Error status
  - External inputs and outputs
  - States (dwork) and initial condition of states
- *rt\_OneStep*: Called at every timer interrupt, implemented in *ert\_main.c*, and contains a call to *model\_step* function, which in turn implements one time step of the model. The call to *rt\_OneStep* is commented out by default and must be attached to a timer or interrupt-service routine with a period that corresponds to the model's sample time.
- *model\_terminate*: Called at the end of the execution, implemented in *model.c*, and contains any necessary termination tasks.



# Code modules

## The Model Step Function

The `model_step` function implements one time step of the model, and is called by the execution harness at every timer interrupt.

```

33 void iir_step(void)
34 {
35     real_T rtb_y;
36
37     /* Sum: '<S2>/Sum' incorporates:
38      * Gain: '<S2>/a1'
39      * Gain: '<S2>/b0'
40      * Inport: '<Root>/u'
41      * UnitDelay: '<S2>/Unit_Delay'
42      */
43     rtb_y = 0.1 * iir_U.u + 0.9 * iir_DW.UnitDelay_DSTATE;
44
45     /* Outport: '<Root>/y' */
46     iir_Y.y = rtb_y;
47
48     /* Update for UnitDelay: '<S2>/Unit_Delay' */
49     iir_DW.UnitDelay_DSTATE = rtb_y;
50 }

```

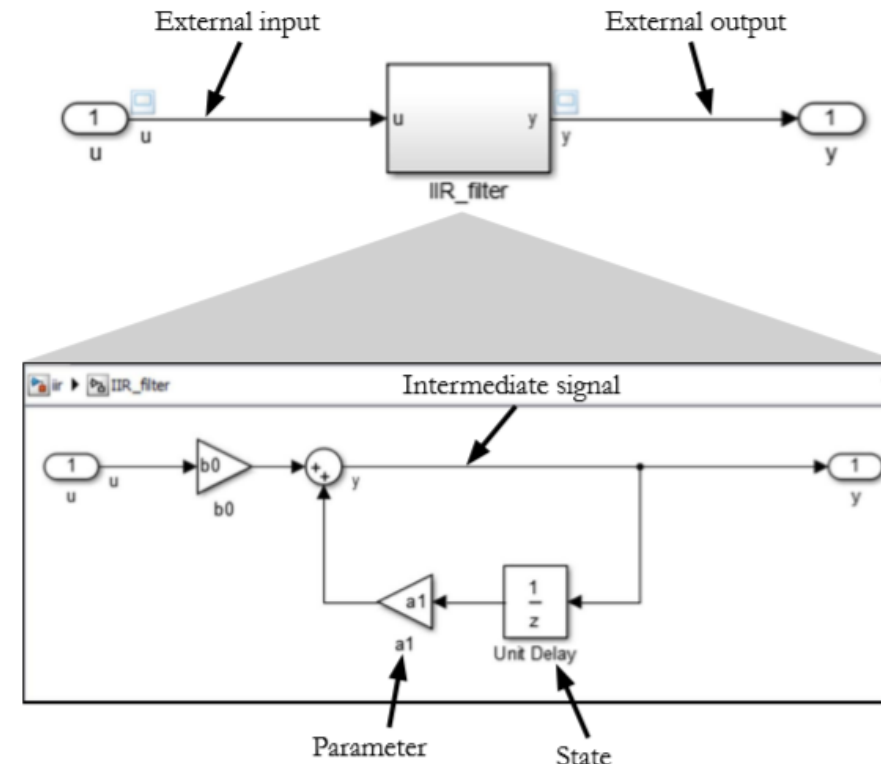
$$y[k] = b_0 \cdot x[k] + a_1 \cdot y[k-1]$$

# Logging intermediate signals

## Simulink Data in Generated Code

Simulink data consists of the following:

- External inputs
  - Signals fed from model-level inports
  - Packed in external inputs structure
- External outputs
  - Signals fed to model-level outports
  - Packed in external outputs structure
- Intermediate signals
  - Signals that flow between two intermediate blocks
  - Packed in model block signals structure
  - Defined as local variables and reused, or reduced if possible
- Discrete states (delays) and shared memory (data store memory)
  - Persistent memory whose values must be carried over from one update to the next as history elements
  - Packed in model block states structure
- Parameters
  - Constant attributes that govern block behavior
  - Packed in model block parameters structure
- Timing information
  - Overrun flag and counters for scheduler use
  - Packed in model object

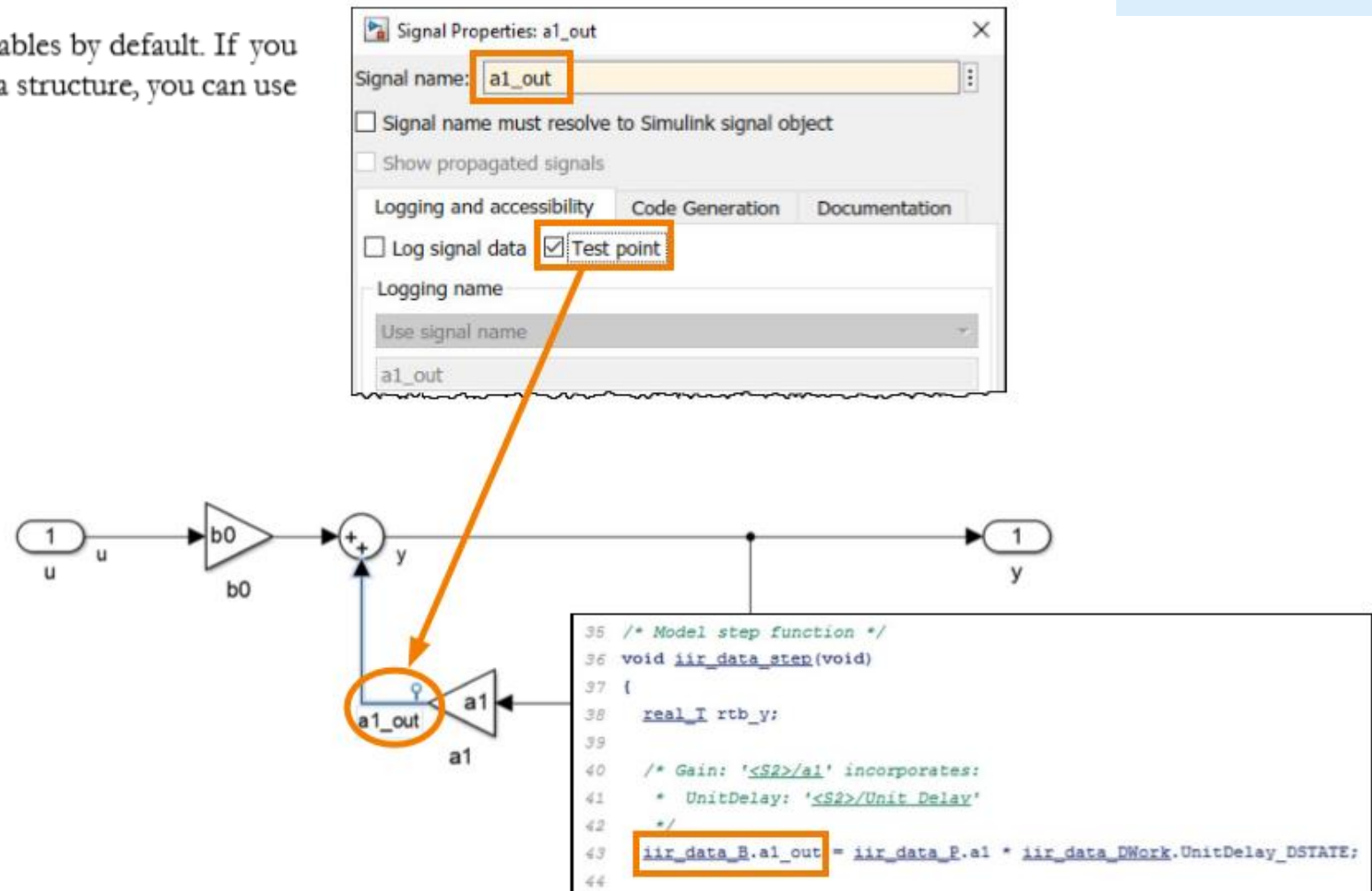


# Logging intermediate signals

Intermediate signals in a model are treated as local variables by default. If you want to move an intermediate signal into the global data structure, you can use test points.

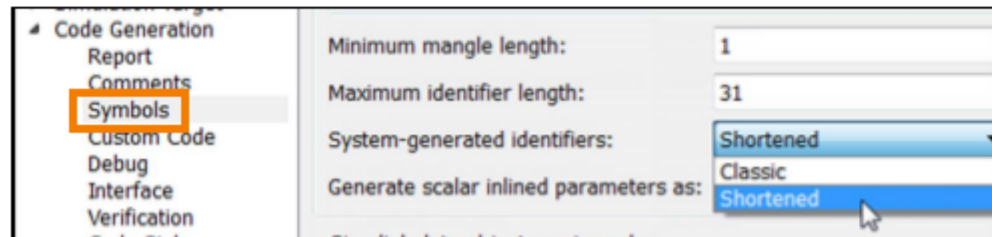
```
>> iir
```

```
>> iir_data
```



# Data structures in generated code

	Classic		Shortened	
	Data type naming	Variable name	Data type naming	Variable name
Model	RT_MODEL_model	model_M	RT_MODEL_model	model_M
Parameters	Parameters_model	model_P	P_model	model_P
External Inputs	ExternalInputs_model	model_U	ExtU_model	model_U
External Outputs	ExternalOutputs_model	model_Y	ExtY_model	model_Y
Block Signals	BlockIO_model	model_B	B_model	model_B
Block States	D_Work_model	model_DWork	DW_model	model_DW

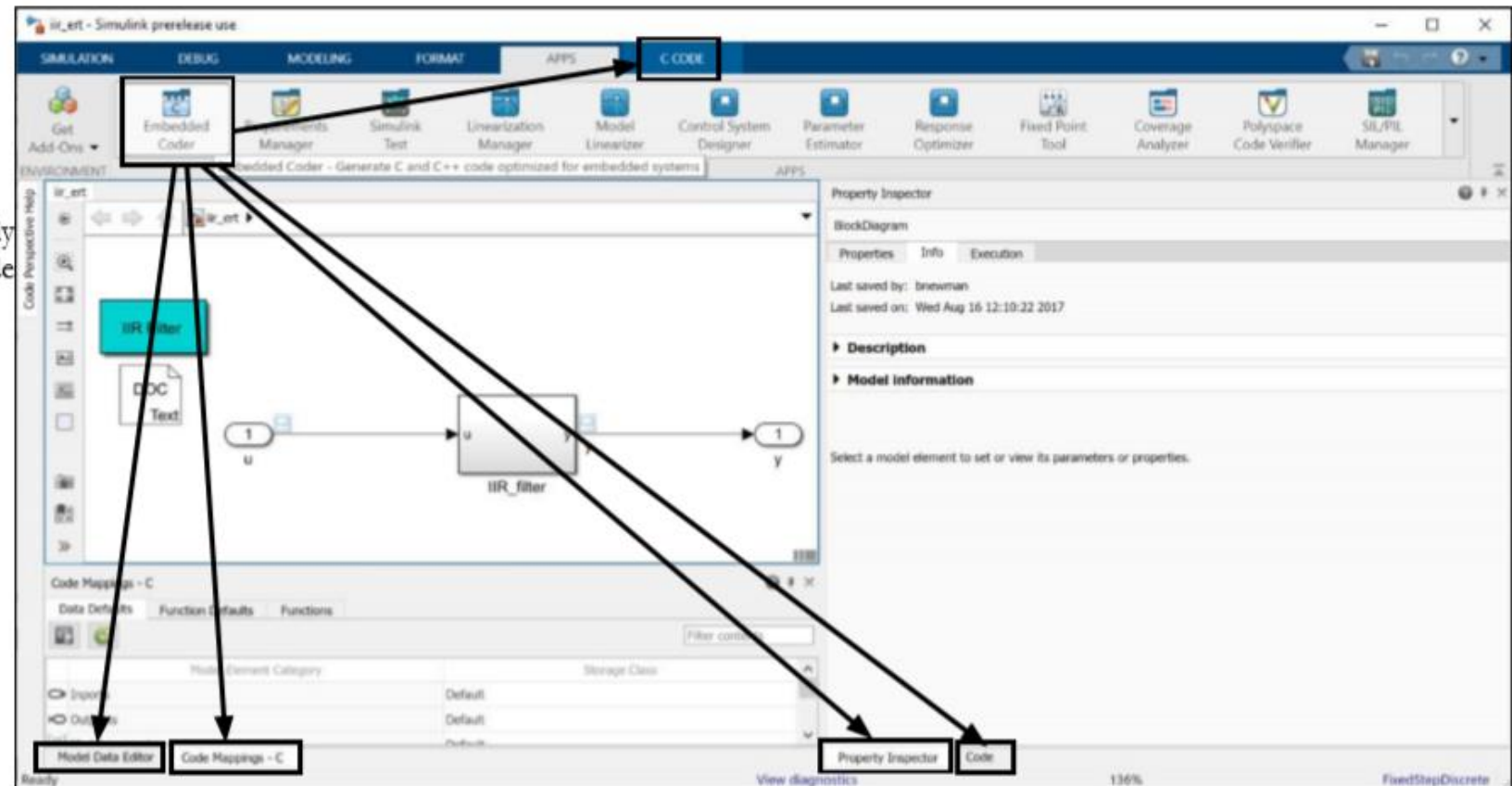


```

19  /* Block state (auto storage) */
20  DW_iir_T iir_DW;
21
22  /* External inputs (root inport signals)
23  ExtU_iir_T iir_U;
24
25  /* External outputs (root outports fed
26  ExtY_iir_T iir_Y;
  
```

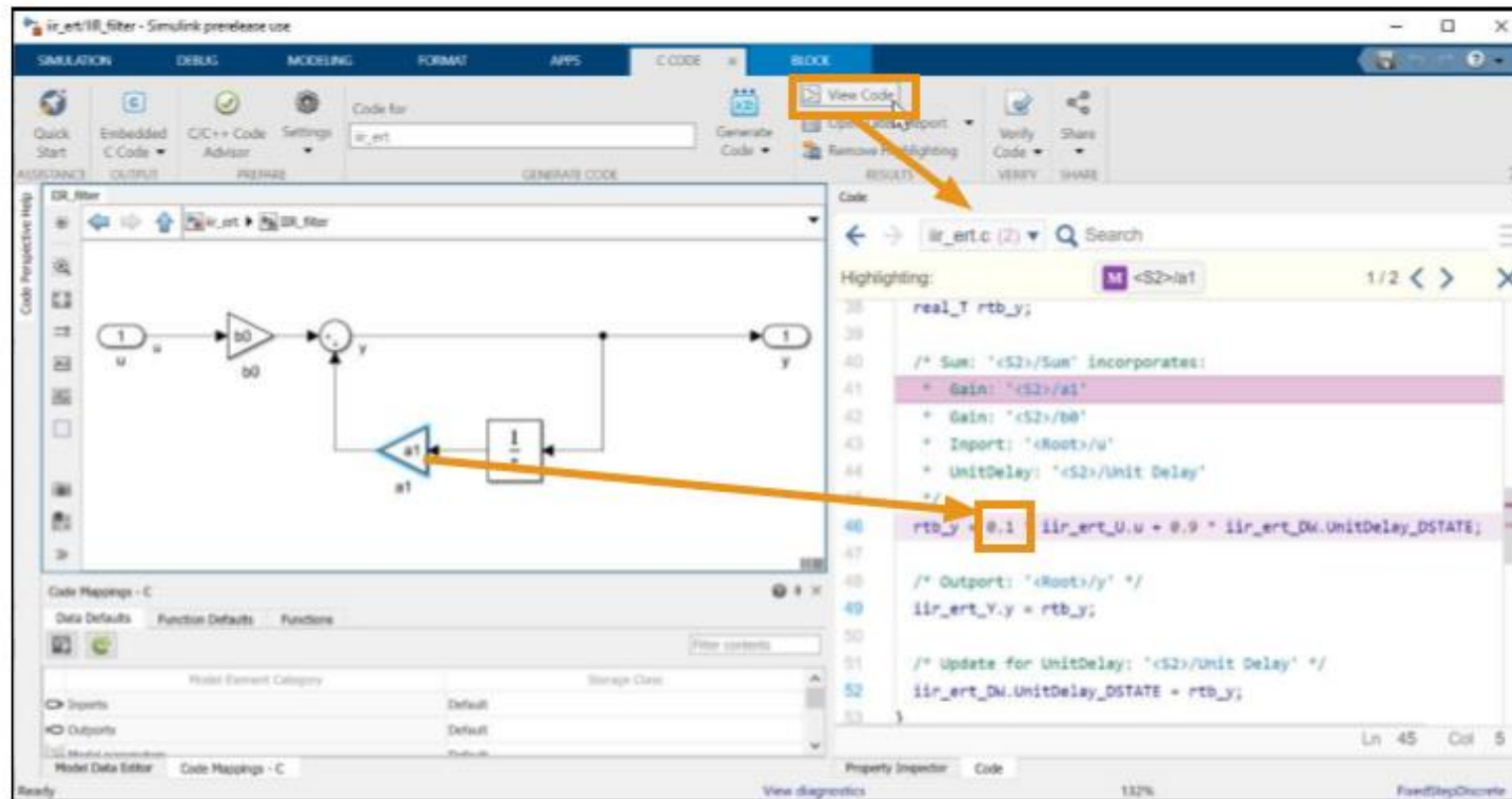
# Embedded Coder App

- **Model Data Editor** – Setting design and code generation properties for individual pieces of data in the model
- **Code Mappings – C** – Setting default behaviors for data categories and generated functions
- **Property Inspector** – Setting block and signal properties
- **Code** – Examining the generated code and its links to model elements
- **C Code** (toolstrip tab) – Interactively performing actions related to code generation and testing





# Code Panel



The screenshot shows the MATLAB/Simulink environment with the 'iir\_ert' block open. The 'C CODE' tab is selected, and the 'View Code' button is highlighted. The C code is displayed in the 'Code' panel, showing the implementation of the block's logic. An orange arrow points from the 'a1' gain block in the Simulink diagram to the corresponding line in the C code.

```

38 real_T rtb_y;
39
40 /* Sum: '<S2>/Sum' incorporates:
41  * Gain: '<S2>/a1'
42  * Gain: '<S2>/b0'
43  * Input: '<Root>/u'
44  * UnitDelay: '<S2>/Unit Delay'
45  */
46 rtb_y = 0.1 iir_ert_U.u + 0.9 * iir_ert_DW.UnitDelay_DSTATE;
47
48 /* Output: '<Root>/y' */
49 iir_ert_Y.y = rtb_y;
50
51 /* Update for UnitDelay: '<S2>/Unit Delay' */
52 iir_ert_DW.UnitDelay_DSTATE = rtb_y;
53 }

```




This close-up shows the C code with the gain block '<S2>/a1' highlighted. The code snippet is:

```

44 * Gain: '<S2>/b0'
45 * Input: '<Root>/u'
46 * UnitDelay: '<S2>/Unit Delay'
47
48 rtb_y = 11r_P.b0 * 11r_U.u + 11r_B.a1_out;
49 }

```



This close-up shows the C code with the variable 'ExtV\_11r\_T 11r\_Y;' highlighted. The code snippet is:

```

44 ExtV_11r_T 11r_Y;
45
46 /* Sum: '<S2>/Sum' incorporates:
47  * Gain: '<S2>/a1'
48  * Gain: '<S2>/b0'
49  * Input: '<Root>/u'
50  * UnitDelay: '<S2>/Unit Delay'
51  */
52 rtb_y = 11r_P.b0 * 11r_U.u + 11r_B.a1_out;
53 }

```



This close-up shows the C code with the variable 'ExtV\_11r\_T 11r\_Y;' highlighted. The code snippet is:

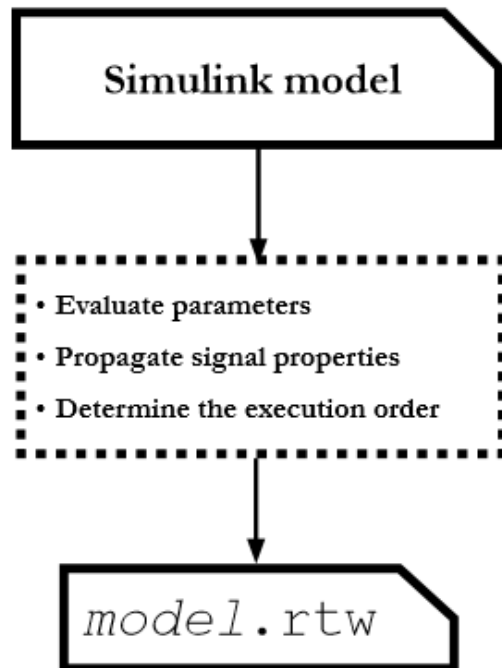
```

24
25 /* External inputs (root input signals with
26 ExtU_11r_T 11r_U;
27
28 /* External outputs (root outputs fed by s
29 ExtV_11r_T 11r_Y;

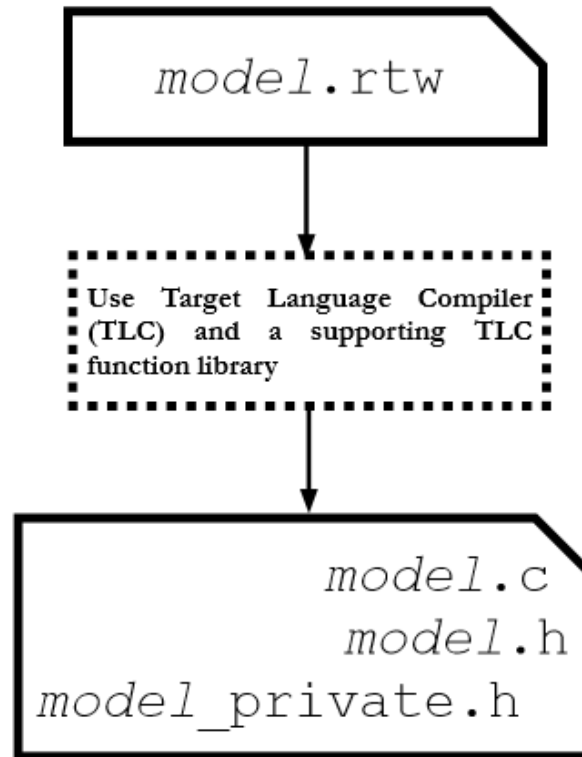
```

# Embedded Coder build process

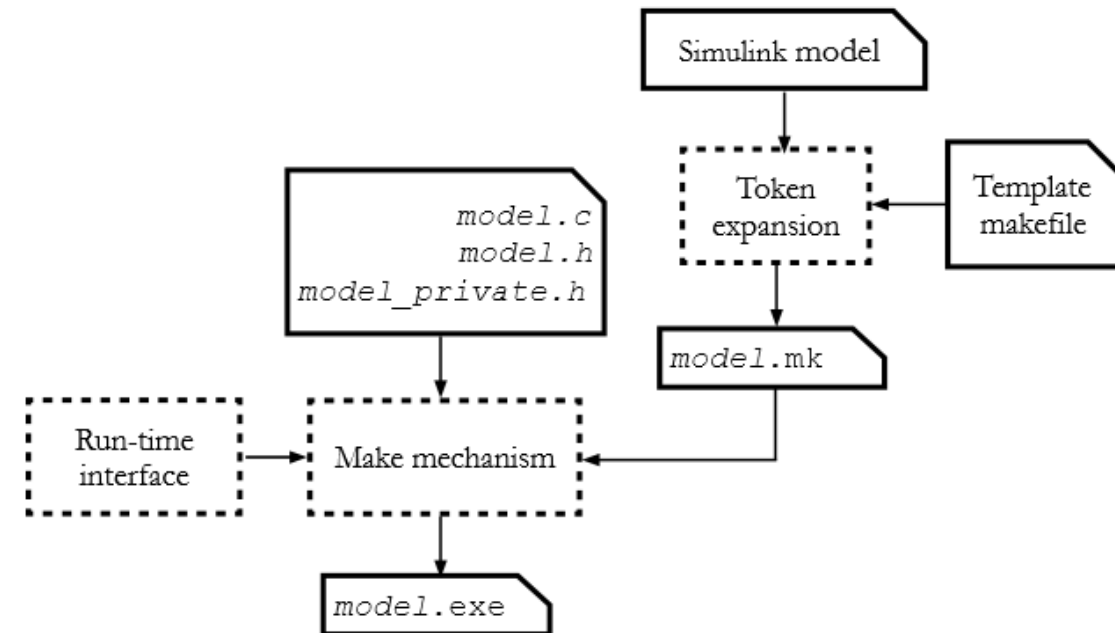
## Model Compilation



## Code Generation



## Executable Generation



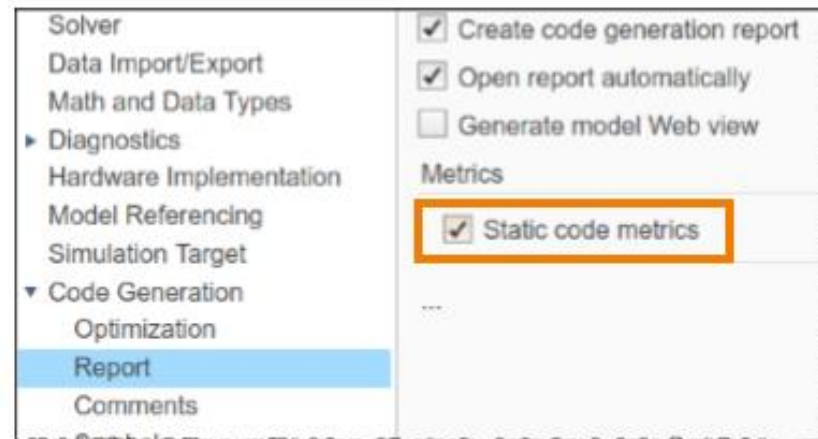
# Optimization considerations

- **Debugging:** Optimize code generation settings to debug the code generation build process.
- **Traceability:** Optimize code generation settings to provide mapping between model elements and code.
- **Efficiency:** Optimize code generation settings to reduce RAM, ROM, and execution time.
- **Safety precaution:** Optimize code generation settings to provide least possibility of errors.

# Static Code Metrics

>> `optim_example`

To evaluate various optimization metrics of the generated code and observe how optimization options affect the code, you can use the static code metrics report.



Contents

- Summary
- Subsystem Report
- Code Interface Report
- Traceability Report
- Static Code Metrics Report**
- Code Replacements Report

Generated Code

- [-] Main file
  - [ert\\_main.c](#)
- [-] Model files
  - [optim\\_example.c](#)
  - [optim\\_example.h](#)
  - [optim\\_example\\_private.h](#)
  - [optim\\_example\\_types.h](#)
- [-] Data files
  - [optim\\_example\\_data.c](#)
- [+] Utility files (1)

## Static Code Metrics Report

The static code metrics report provides statistics of the generated code. Metrics are estimated from static analysis of the generated code using the C data types specified in the **Hardware Implementation > Embedded hardware** pane of the Configuration Parameters dialog box: **char** 8, **short** 16, **int** 32, **long** 32, **float** 32, **double** 64, **pointer** 32 bits. Actual object code metrics might differ due to target specific compiler and platform settings. Consult the **Code Generation Advisor** for options to improve code efficiency.

### Table of Contents

- [File Information](#)
- [Global Variables](#)
- [Function Information](#)

#### 1. File Information [hide]

[-] Summary (excludes ert\_main.c)

Number of .c files: 2  
Number of .h files: 4  
Lines of code: 195  
Lines: 491

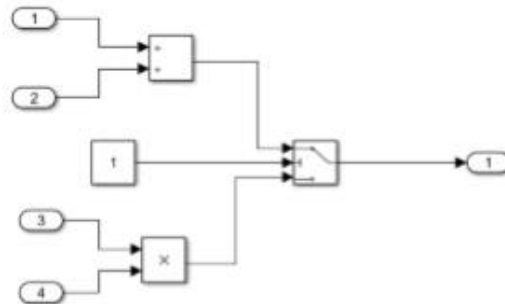
[-] File details

File Name	Lines of Code	Lines	Generated On
<a href="#">rtwtypes.h</a>	103	190	12/03/2014 1:46 PM
<a href="#">optim_example.h</a>	43	112	12/03/2014 1:46 PM
<a href="#">optim_example.c</a>	33	96	12/03/2014 1:46 PM
<a href="#">optim_example_data.c</a>	6	34	12/03/2014 1:46 PM
<a href="#">optim_example_types.h</a>	6	33	12/03/2014 1:46 PM
<a href="#">optim_example_private.h</a>	4	26	12/03/2014 1:46 PM

# Removing unnecessary code

One class of code optimizations is removing unnecessary code. This includes the code generated for the following:

- Data initialization
- Termination
- Unnecessary blocks



optim\_example.c

```
32 /* Model step function */
33 void optim_example_step(void)
34 {
35     /* Sum: 'sRoots/Add' incorporates:
36      * Input: 'sRoots/In1'
37      * Input: 'sRoots/In2'
38      */
39     optim_example_R.Add = optim_example_U.In1 + optim_example_U.In2;
40
41     /* Product: 'sRoots/Product' incorporates:
42      * Input: 'sRoots/In3'
43      * Input: 'sRoots/In4'
44      */
45     optim_example_R.Product = optim_example_U.In3 * optim_example_U.In4;
46
47     /* Switch: 'sRoots/Switch' incorporates:
48      * Constant: 'sRoots/Constant'
49      */
50     if (optim_example_P.Constant_Value >= optim_example_P.Switch_Threshold
51         /* Outport: 'sRoots/Out1' */
52         optim_example_Y.Out1 = optim_example_R.Add;
53     ) else {
54         /* Outport: 'sRoots/Out1' */
55         optim_example_Y.Out1 = optim_example_R.Product;
56     }
```

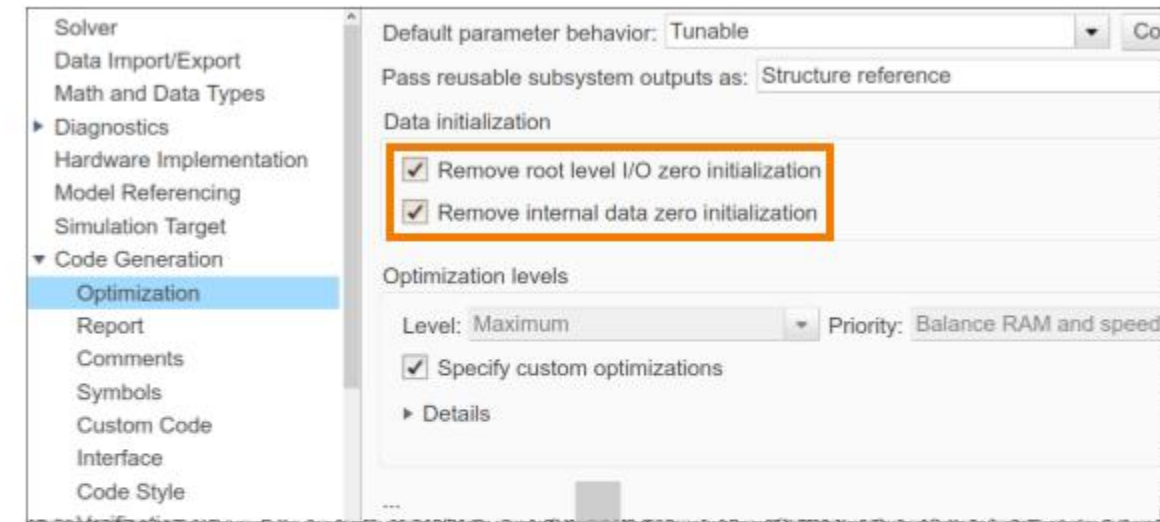
optim\_example.c

```
61 /* Model initialize function */
62 void optim_example_initialize(void)
63 {
64     /* Registration code */
65
66     /* initialize error status */
67     rtmSetErrorStatus(optim_example_M, (NULL));
68
69     /* block I/O */
70     {
71         optim_example_R.Add = 0.0;
72         optim_example_R.Product = 0.0;
73     }
74
75     /* external inputs */
76     optim_example_U.In1 = 0.0;
77     optim_example_U.In2 = 0.0;
78     optim_example_U.In3 = 0.0;
79     optim_example_U.In4 = 0.0;
80
81     /* external outputs */
82     optim_example_Y.Out1 = 0.0;
83 }
84
85 /* Model terminate function */
86 void optim_example_terminate(void)
87 {
88     /* (no terminate code required) */
89 }
```

# Removing unnecessary code

## Removing Initialization Code

Initializing the internal and external data whose value is zero is a precaution and may not be necessary for your application. Many embedded application environments initialize all RAM to zero at startup, making generation of initialization code redundant.



```

61  /* Model initialize function */
62  void optim_example_initialize(void)
63  {
64      /* (no initialization code required) */
65  }
    
```



# Removing unnecessary code

## Removing Initialization Code

A related option, **Use memset to initialize floats and doubles to 0** (located in **Code Generation > Optimization** under **Advanced parameters**), lets you control the representation of zero used during initialization. When this option is enabled, `memset` is used to clear internal storage for floating-point data to integer bit pattern 0 (all bits 0), regardless of type. When it is disabled, extra code is generated to explicitly initialize storage for data of types `float` and `double` to 0.0. The resulting code is slightly less efficient than code generated when you select the option.

These options have no impact on debugging or traceability. Set them for efficiency and clear them for safety.

Enabling this option in Configuration Parameters will have:

✓: positive effect

✗: negative effect

–: no effect

Debugging	–
Traceability	–
Efficiency	✓
Safety	✗
Default	Off

☐ Use memset to initialize floats and doubles to 0.0

Function Name	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
<a href="#">optim_example_initialize</a>	0	0	10	22	1
<a href="#">optim_example_step</a>	0	0	7	27	2
<a href="#">optim_example_terminate</a>	0	0	0	4	1

☒ Use memset to initialize floats and doubles to 0.0

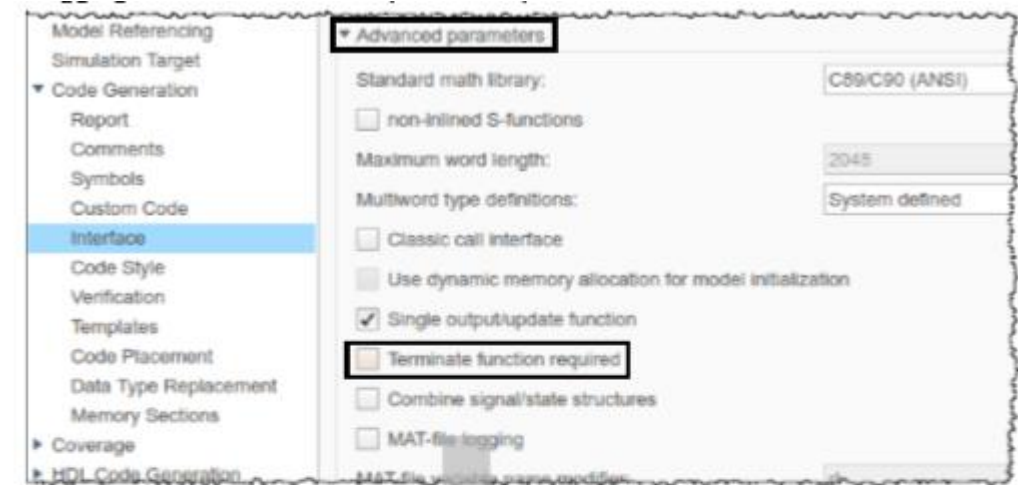
Function Name	Accumulated Stack Size (bytes)	Self Stack Size (bytes)	Lines of Code	Lines	Complexity
<a href="#">optim_example_initialize</a>	0	0	0	4	1
<a href="#">optim_example_step</a>	0	0	7	27	2
<a href="#">optim_example_terminate</a>	0	0	0	4	1

# Removing unnecessary code

## Removing Termination Code

`model_terminate` contains all model termination code and should be called as part of system shutdown. If your application runs indefinitely, you do not need the `model_terminate` function. To suppress the function, open Configuration Parameters, go to the **Code Generation > Interface** pane, and clear the **Terminate function required** option. You need to click **Advanced parameters** to see this option.

This option has no impact on debugging. Clear it for efficiency and safety.



Debugging	—
Traceability	—
Efficiency	✖
Safety	✖
Default	On

```

61  /* Model initialize function */
62  void optim_example_initialize(void)
63  {
64      /* (no initialization code required) */
65  }
66  .....
67  /*
68  * File trailer for generated code.

```

# Removing unnecessary data support

The **Interface** subpane under **Code Generation** pane allows you to modify the following options:

## Support floating-point/non-finite/complex numbers

These options let you enable or suppress the generation of floating-point, nonfinite, or complex numbers. By default, all three options are selected.

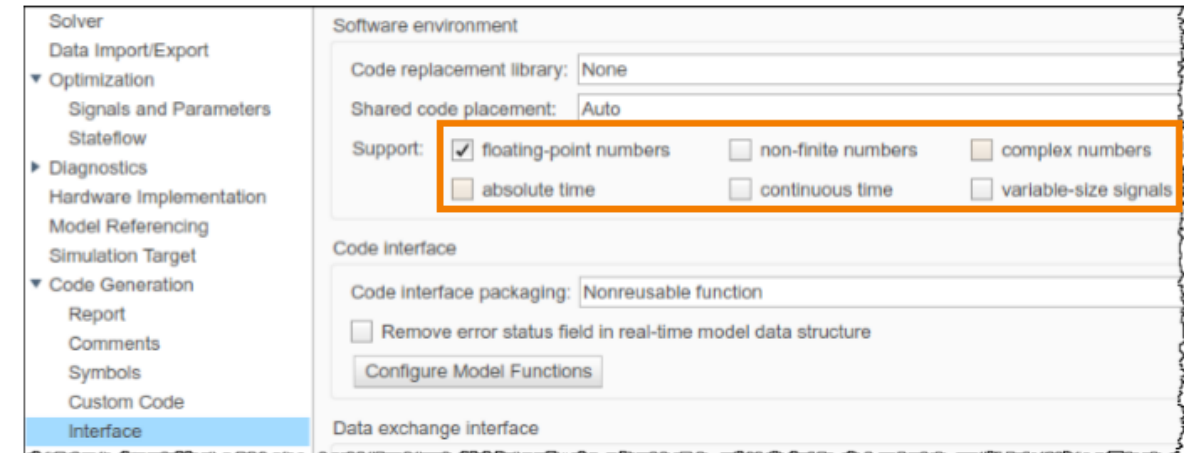
To generate pure integer code, deselect the **Support floating-point numbers** option. If your model requires generation of floating-point data or operations, select the option.

The **Support non-finite numbers** option is enabled only when Support floating-point numbers is selected. This option lets you enable or suppress generation of non-finite values.

The **Support complex numbers** option is independent of the other two options. This option lets you enable or suppress generation of complex numbers.

## Support absolute time

Certain blocks require the value of either absolute time or elapsed time (for example, the time elapsed between two trigger events). These related options determine how the ERT target provides absolute or elapsed time values to blocks in the model.



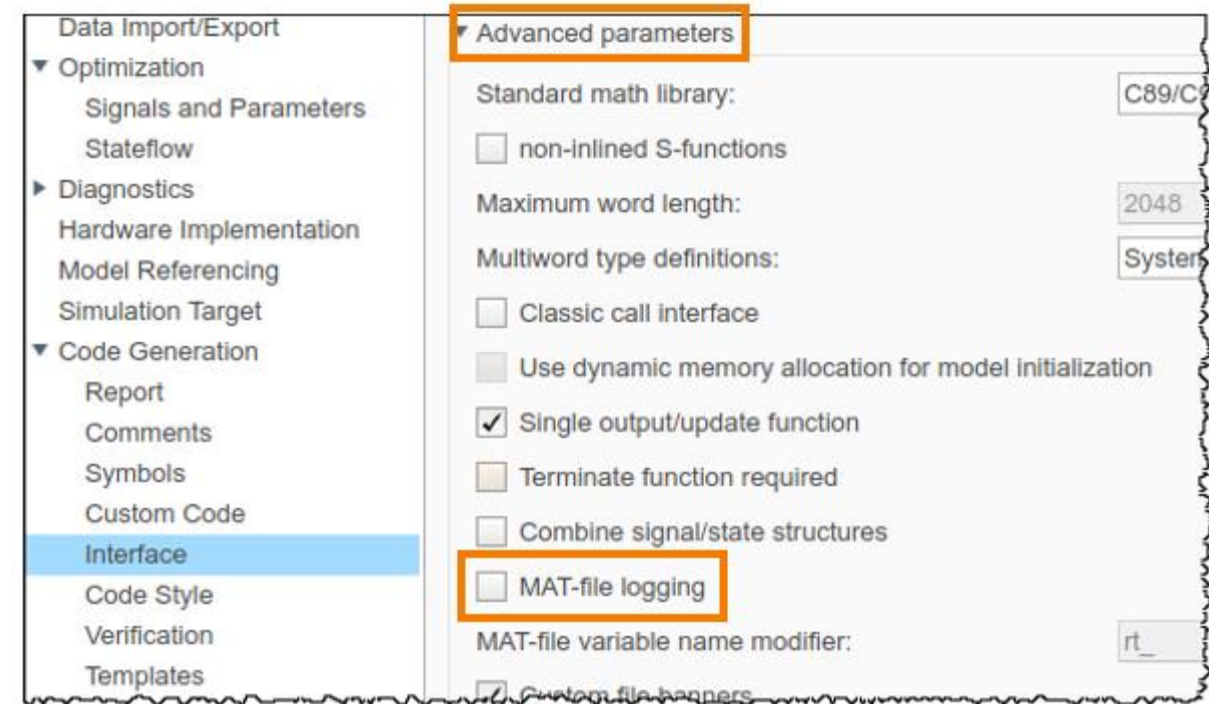
Debugging	–
Traceability	–
Efficiency	✓
Safety	✓
Default	Vary

## Support continuous time

If this option is selected, the ERT target supports code generation for continuous-time blocks. By default, this option is deselected, and the build process generates an error if any continuous-time blocks are present in the model.

# MAT-File Logging

The **MAT-file logging** option controls logging of signals. For embedded applications, which typically do not support a file system, you must clear this option. Suppression of MAT-file logging eliminates the extra code and memory usage required to initialize, update, and clean up logging variables.



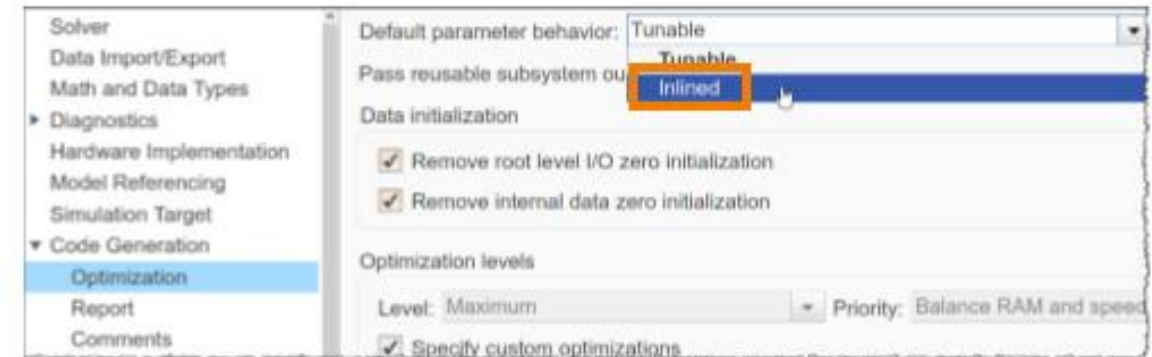
Debugging	✓
Traceability	–
Efficiency	✖
Safety	✖
Default	Off



# Inlined Parameters

The **Default parameter behavior** option controls how numeric block parameters appear in code generated from the model. When this optimization is set to **Tunable**, a model's mathematical block parameters appear as variables in the generated code. As a result, you can tune the parameters when executing the code.

When the **Default parameter behavior** is set to **Inlined** (default), block parameters are evaluated at code generation time, and their values appear as constants in the generated code, if possible. This reduces the generated code's memory and processing requirements. However, because the inlined parameters appear as constants in the generated code, you cannot tune them during code execution.



```
29 /* Model step function */
30 void optim_example_step(void)
31 {
32     /* Switch: '<Root>/Switch' incorporates:
33      * Constant: '<Root>/Constant'
34      */
35     if (optim_example_P.Constant_Value >= optim_example_P.Switch_Threshold) {
36         /* Output: '<Root>/Out1' incorporates:
37          * Input: '<Root>/In1'
38          * Input: '<Root>/In2'
39          * Sum: '<Root>/Add'
40          */
41     }
```



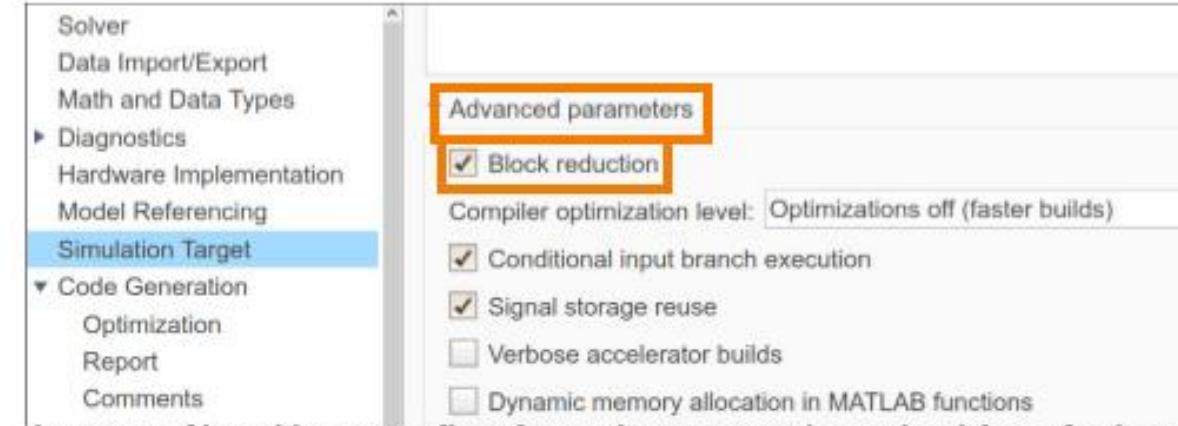
```
29 /* Model step function */
30 void optim_example_step(void)
31 {
32     /* Switch: '<Root>/Switch' incorporates:
33      * Constant: '<Root>/Constant'
34      */
35     if (1.0 >= 0.0) {
36         /* Output: '<Root>/Out1' incorporates:
37          * Input: '<Root>/In1'
38          * Input: '<Root>/In2'
39          * Sum: '<Root>/Add'
40          */
41     }
```

Debugging	✗
Traceability	✗
Efficiency	✓
Safety	–
Default	Off

# Block Reduction

The three types of block reduction are:

- **Removal of Redundant Type Conversions:** Unnecessary type conversion blocks are removed. For example, an int type conversion block whose input and output are of type int is redundant and is removed.
- **Dead Code Elimination:** Any blocks or signals in an unused code path are eliminated from generated code. This includes blocks whose output signal paths terminate at a non-executing block, such as a Terminator block or a disabled Assertion block, provided none of the signals require global storage.
- **Removal of Fast-to-Slow Rate Transition Blocks in a Single-Tasking System:** In this scenario, where the input signal sampling frequency is higher than the output signal sampling frequency, the Rate Transition block does not implement any operations.



```

29 /* Model step function */
30 void optim_example_step(void)
31 {
32     /* Outport: '<Root>/Out1' incorporates:
33      * Inport: '<Root>/In1'
34      * Inport: '<Root>/In2'
35      * Sum: '<Root>/Add'
36      */
37     optim_example_Y.Out1 = optim_example_U.In1 + optim_example_U.In2;
38 }

```

Debugging	✗
Traceability	– ✗
Efficiency	✓
Safety	✗
Default	On

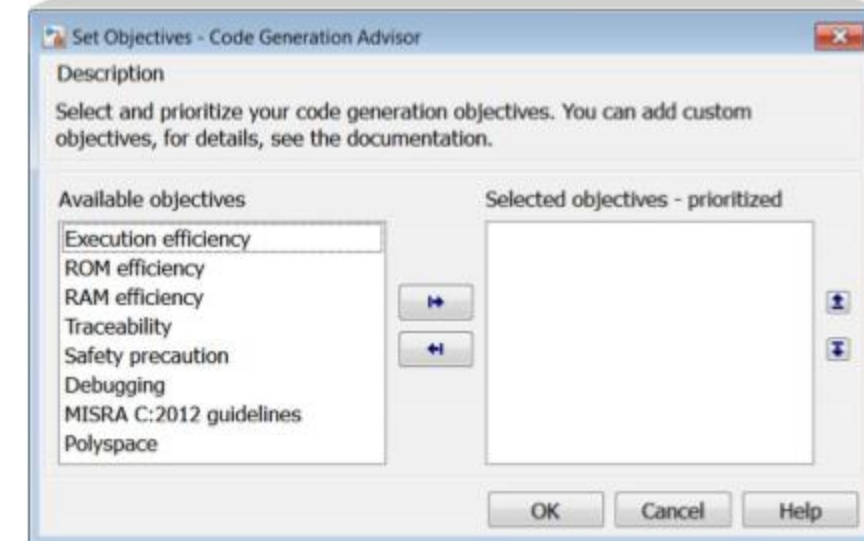


# Code Generation Objectives

The Embedded Coder product allows you to check the optimization settings of your model and change them collectively.

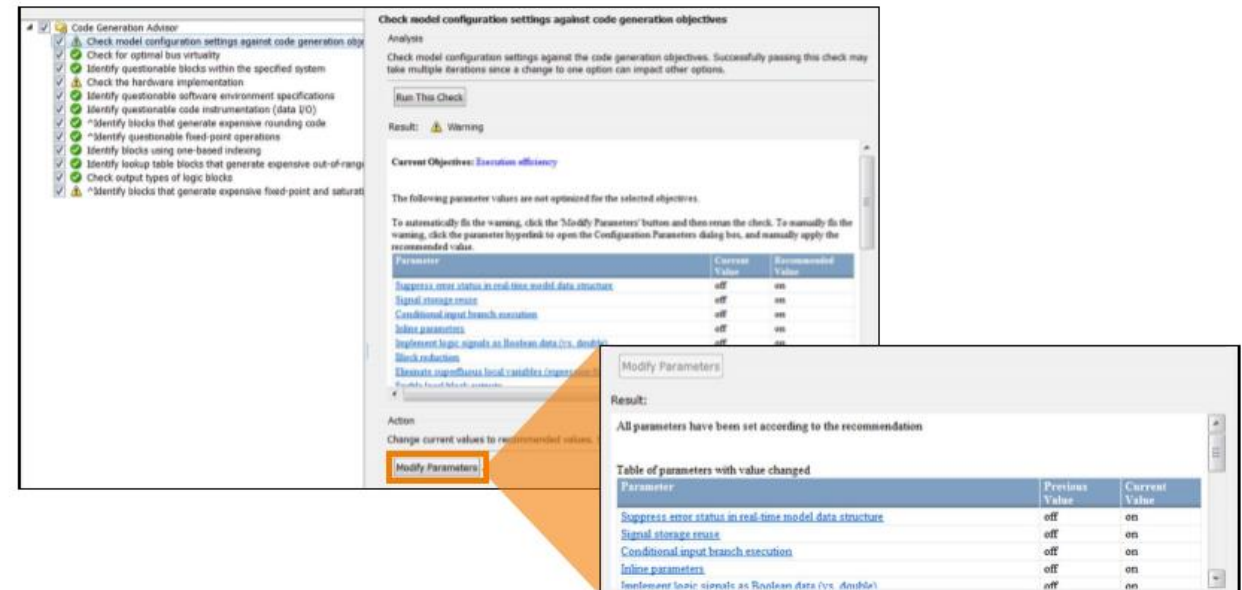
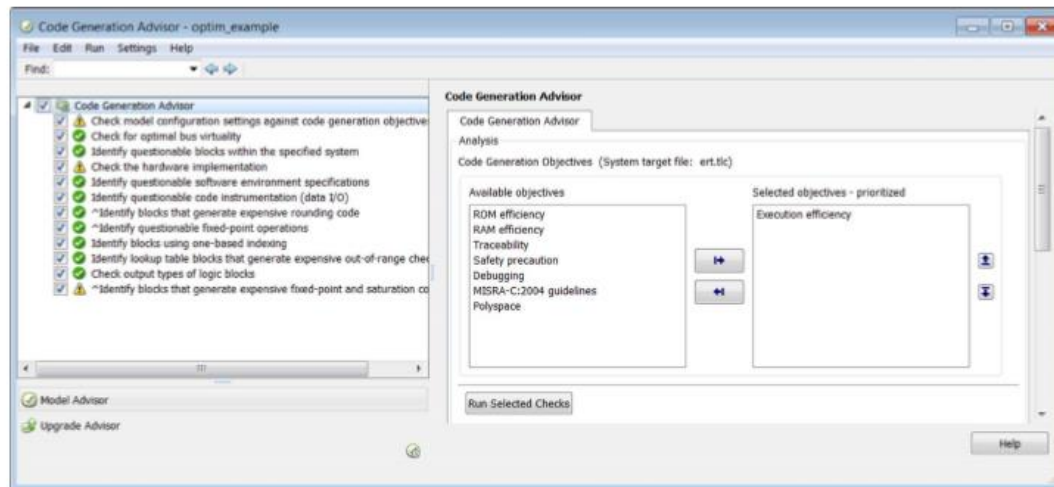
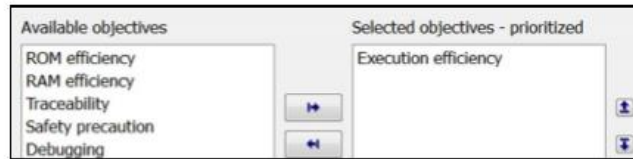
The available objectives are

- Execution Efficiency
- ROM Efficiency
- RAM Efficiency
- Traceability
- Safety precaution
- Debugging
- MISRA-C:2012 guidelines
- Polyspace



# Code Generation Objectives

```
>> optim_example
```



# Embedded Coder Quick Start

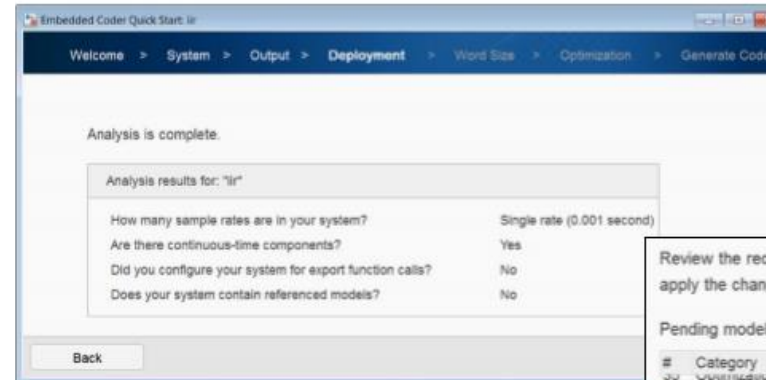
>> iir

The Quick Start Tool:

- Asks a few questions about your code generation goals and your target hardware.
- Validates your model against your selections.
- Shows you the recommended configuration changes.
- Applies the configuration changes and generates code.

The Quick Start Tool examines your model to determine:

- How many sample rates are in your system.
- If there are continuous-time components.
- If you configured your system for export function calls.
- If your system contains referenced models.



Review the recommended configuration parameter changes for model: 'iir'. When you are ready to apply the changes and generate code, click **Next**.

Pending model configuration parameters changes:

#	Category	Parameter	New Value	Old Value
36	Optimization	Remove root level I/O zero initialization	off	on
37	Optimization	Use bitsets for storing Boolean data	on	off
38	Optimization	Use bitsets for storing state configuration	on	off
39	Optimization	Use division for fixed-point net slope computation	On	Off
40	Optimization	Use memset to initialize floats and doubles to 0.0	off	on
41	Solver	Solver	auto (Automatic solver selection)	discrete (no continuous states)
42	Solver	Tasking mode for periodic sample times	SingleTasking	Auto
43	Solver	Type	Fixed-step	Variable-step

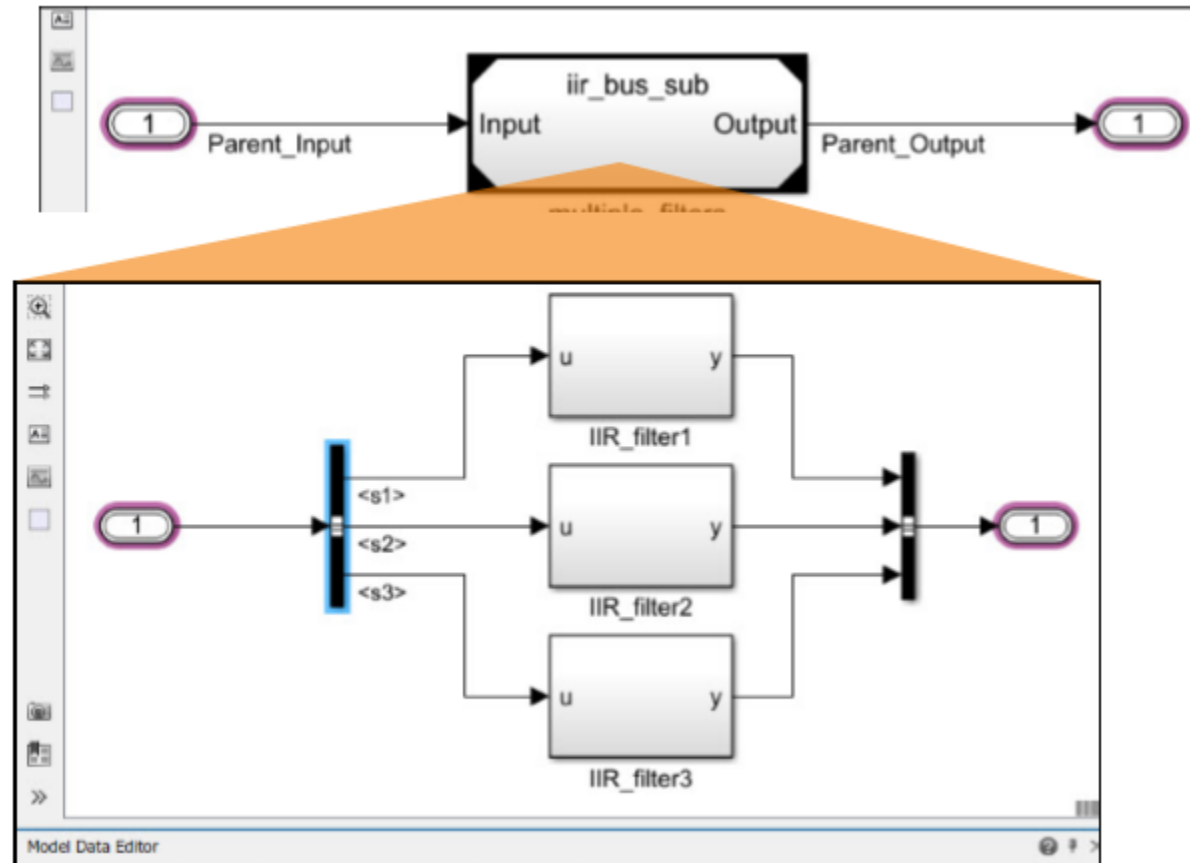
# Bus signals and model referencing

```
>> iir_bus_ref
```

A bus is a composite signal that is implemented as a hierarchical structure. The components of a bus can have different attributes and can themselves be composite signals (buses or vectors).

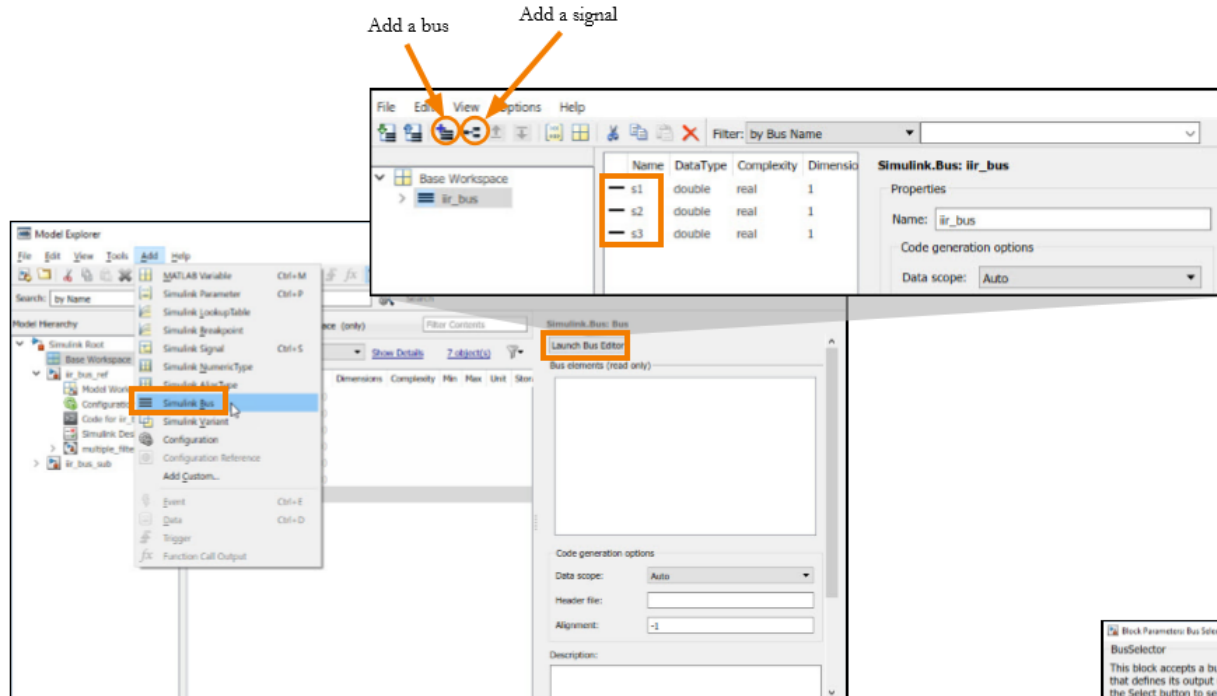
A bus signal allows signals of various data types and dimensions to be propagated on a single signal line. It combines multiple signals into one signal line to simplify routing and allow clean interfacing between models and subsystems.

When using model referencing, bus objects are necessary for propagating signal properties across multiple models that use bus signals.

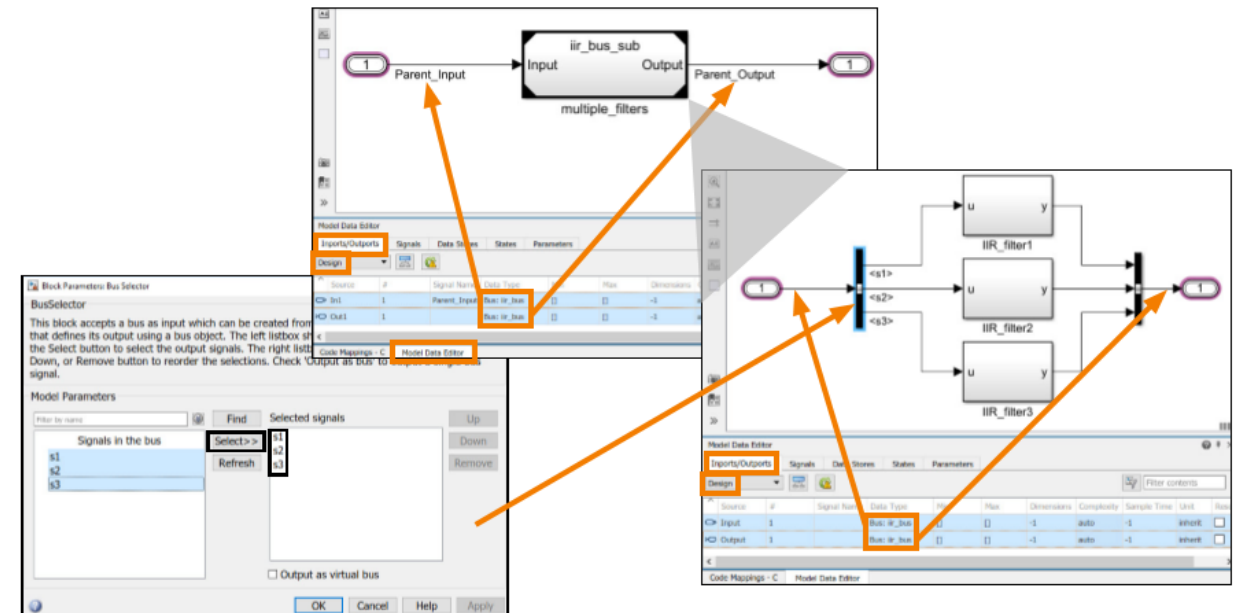


# Bus signals and model referencing

## Creating Bus Objects

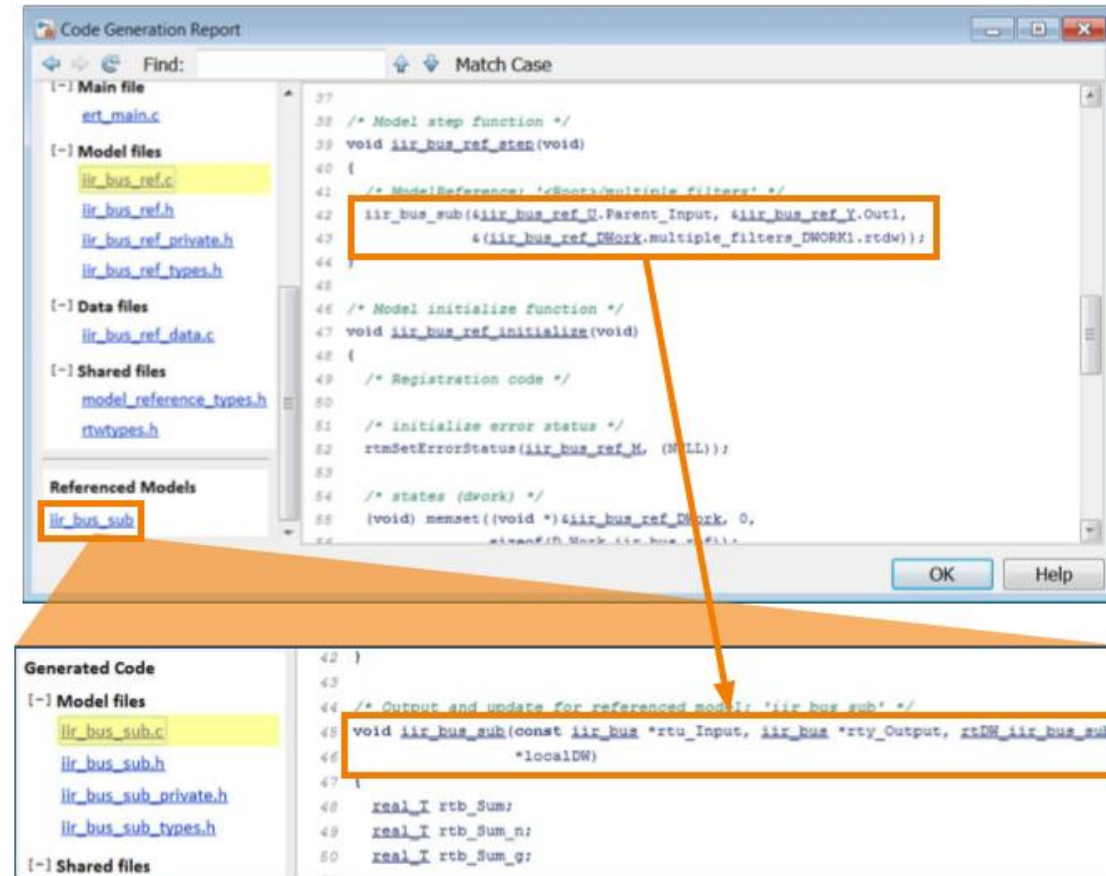


## Assigning Bus Objects



# Model Referencing in Generated Code

The generated code for both parent and referenced models is included in the same code generation report. The `model_step` interface of the referenced model passes its input and output signals as pass-by-reference arguments.



**Code Generation Report**

**Find:** Match Case

**[-] Main file**  
[ert\\_main.c](#)

**[-] Model files**  
[lir\\_bus\\_ref.c](#)  
[lir\\_bus\\_ref.h](#)  
[lir\\_bus\\_ref\\_private.h](#)  
[lir\\_bus\\_ref\\_types.h](#)

**[-] Data files**  
[lir\\_bus\\_ref\\_data.c](#)

**[-] Shared files**  
[model\\_reference\\_types.h](#)  
[rtwtypes.h](#)

**Referenced Models**  
[lir\\_bus\\_sub](#)

```

37
38 /* Model step function */
39 void lir_bus_ref_step(void)
40 {
41     /* ModelReference: 'cSource/multiple_filters' */
42     lir_bus_sub(&lir_bus_ref_U.Parent_Input, &lir_bus_ref_Y.Out1,
43               &lir_bus_ref_DWork.multiple_filters_DWORK1.rtdw);
44 }
45
46 /* Model initialize function */
47 void lir_bus_ref_initialize(void)
48 {
49     /* Registration code */
50
51     /* initialize error status */
52     rtmSetErrorStatus(lir_bus_ref_M, (NLL));
53
54     /* states (dwork) */
55     (void) memset((void *)&lir_bus_ref_DWork, 0,
56                  sizeof(DWork_lir_bus_ref));
57 }

```

**Generated Code**

**[-] Model files**  
[lir\\_bus\\_sub.c](#)  
[lir\\_bus\\_sub.h](#)  
[lir\\_bus\\_sub\\_private.h](#)  
[lir\\_bus\\_sub\\_types.h](#)

**[-] Shared files**

```

42 }
43
44 /* Output and update for referenced model: 'lir_bus_sub' */
45 void lir_bus_sub(const lir_bus *rtu_Input, lir_bus *rty_Output, rtdw_lir_bus_sub
46               *localDW)
47 {
48     real_T rtb_Sum;
49     real_T rtb_Sum_n;
50     real_T rtb_Sum_g;
51 }

```



# Buses in Generated Code

Bus objects are represented by structures in the generated code. Type definition of this structure is done in `model_types.h` of both the parent and the referenced models.

```
iir_bus_ref.h
51 /* External inputs (root inport signals with auto storage) */
52 typedef struct {
53     iir_bus Parent_Input;
54 } ExternalInputs_iir_bus_ref;
55
56 /* External outputs (root outports fed by signals with auto storage) */
57 typedef struct {
58     iir_bus Out1;
59 } ExternalOutputs_iir_bus_ref;
```

```
iir_bus_ref_types.h
iir_bus_sub_types.h
```

```
23 typedef struct {
24     real_T signal1;
25     real_T signal2;
26     real_T signal3;
27 } iir_bus;
```

```
iir_bus_ref.c
```

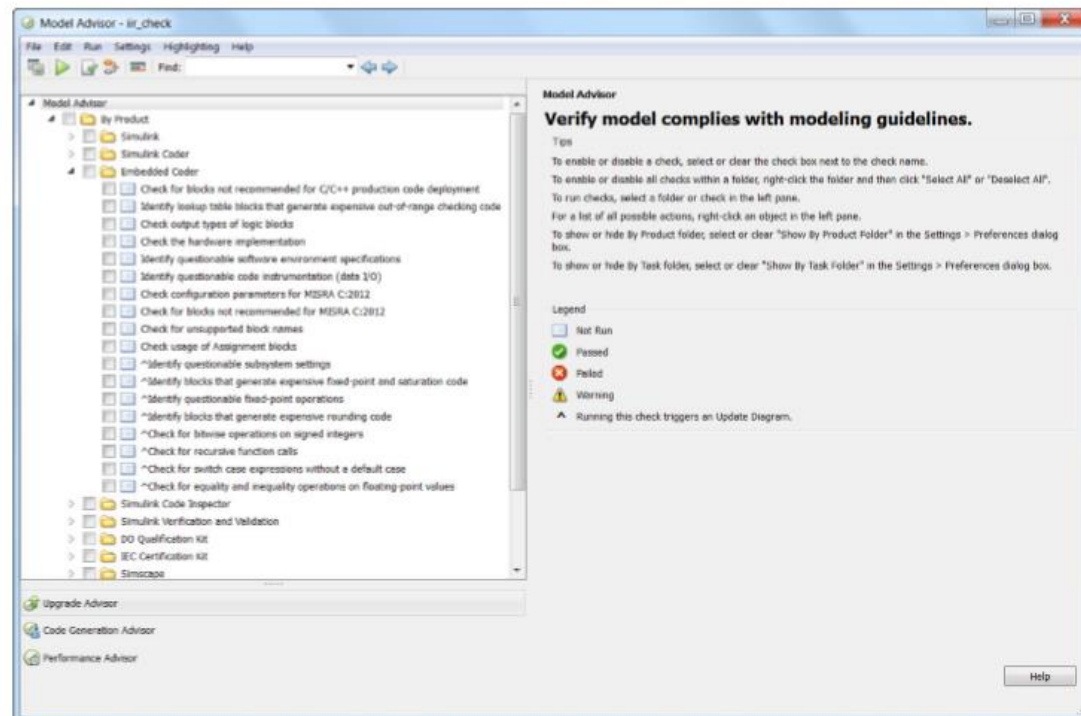
```
28 /* External inputs (root inport signals with auto storage) */
29 ExternalInputs_iir_bus_ref iir_bus_ref_U;
30
31 /* External outputs (root outports fed by signals with auto storage) */
32 ExternalOutputs_iir_bus_ref iir_bus_ref_Y;
33
34 /* Real-time model */
35 RT_MODEL_iir_bus_ref iir_bus_ref_M;
36 RT_MODEL_iir_bus_ref *const iir_bus_ref_M = &iir_bus_ref_M;
37
38 /* Model step function */
39 void iir_bus_ref_step(void)
40 {
41     /* ModelReference: '<Root>/multiple_filters' */
42     iir_bus_sub &iir_bus_ref_U.Parent_Input, &iir_bus_ref_Y.Out1,
43         &(iir_bus_ref_DWork.multiple_filters_DWORK1.rtdw));
44 }
```

```
iir_bus_sub.c
```

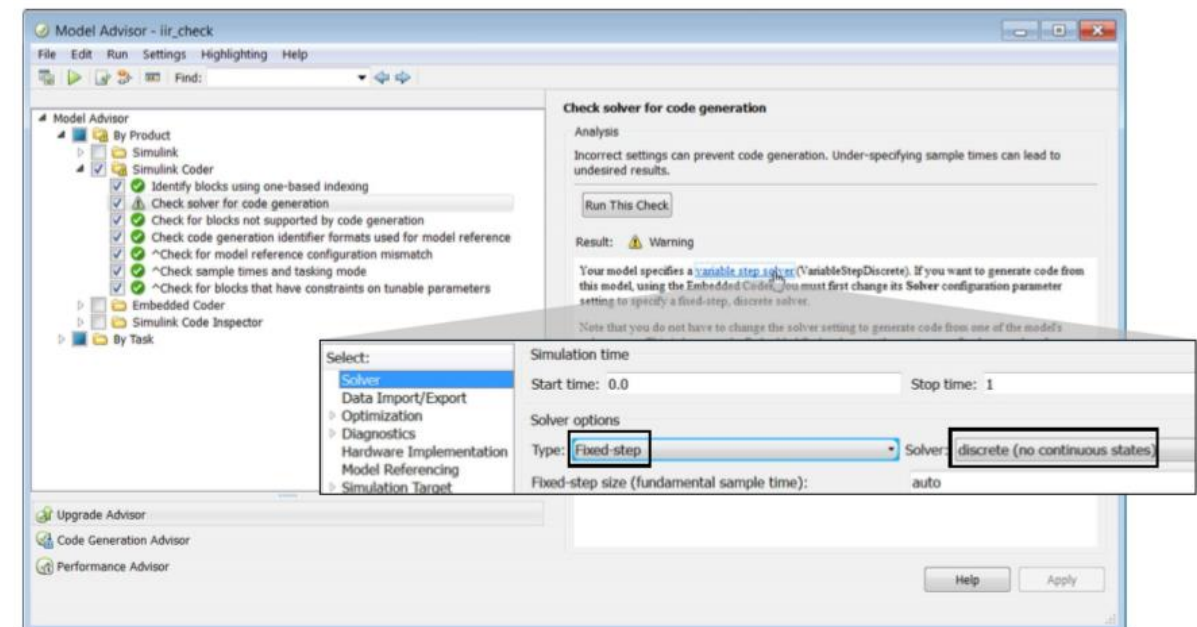
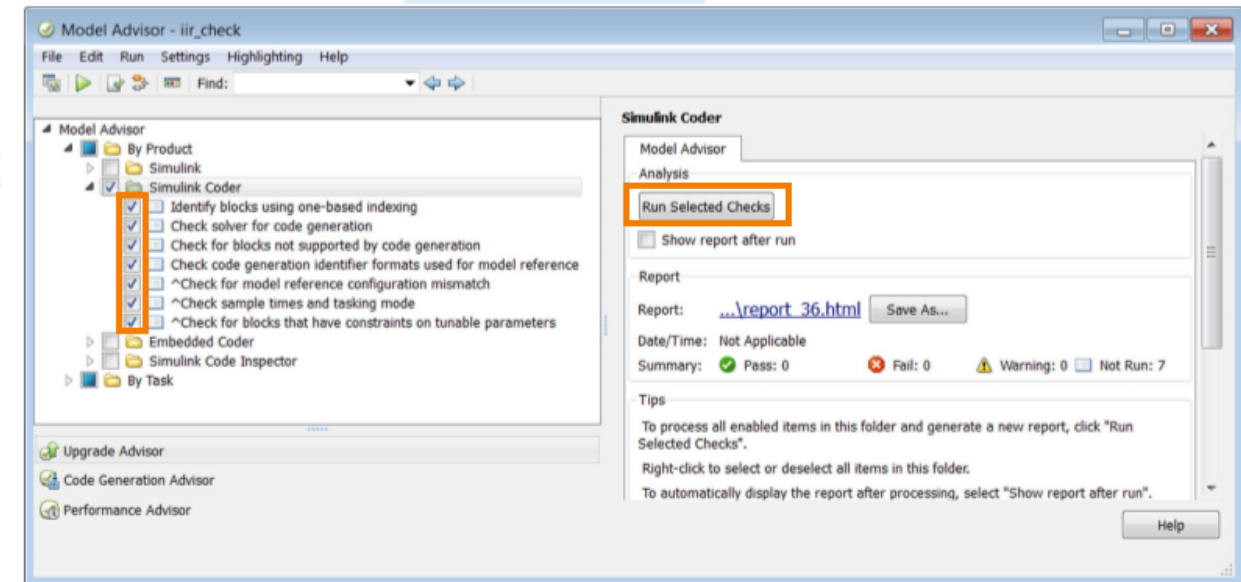
```
44 /* Output and update for referenced model: 'iir bus sub' */
45 void iir_bus_sub(const iir_bus *rtu_Input, iir_bus *rty_Output, rtDW_iir_bus_sub
46     *localDW)
47 {
48     real_T rtb_Sum;
```

# The Model Advisor

The Model Advisor checks a model or subsystem for conditions and configuration settings that can result in inaccurate or inefficient simulation or code generation. It produces a report that lists all the suboptimal conditions or settings that it finds, suggesting better model configuration settings where appropriate.



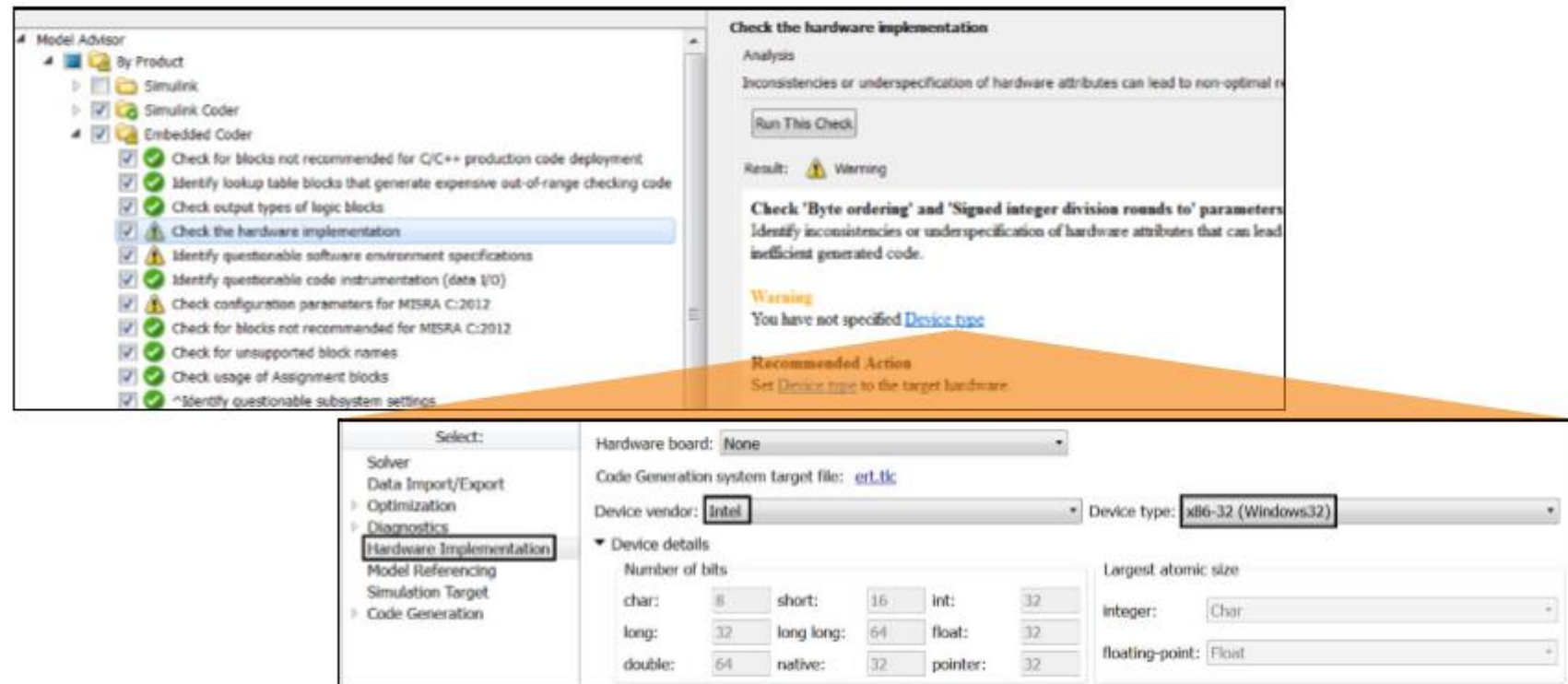
>> iir\_check



# Hardware implementation parameters

After all the checks for **Simulink Coders** are passed, you can move to **Embedded Coder** checks.

The first **Embedded Coder** check to produce a warning is **Check the hardware implementation**. This model does not specify a device type, and this can cause the generated code to be incorrect or inefficient. You can set the hardware implementation from the Hardware Implementation pane of the Configuration Parameters dialog.



The screenshot shows the **Model Advisor** interface with the **Embedded Coder** checks expanded. The **Check the hardware implementation** check is highlighted, showing a warning result. Below this, the **Configuration Parameters** dialog is open, showing the **Hardware Implementation** pane. The **Device vendor** is set to **Intel** and the **Device type** is set to **x86-32 (Windows32)**. The **Device details** section shows the number of bits for various data types.

Number of bits			
char:	8	short:	16
long:	32	long long:	64
double:	64	native:	32
int:	32	float:	32
pointer:	32		

The **Largest atomic size** section shows the integer type set to **Char** and the floating-point type set to **Float**.



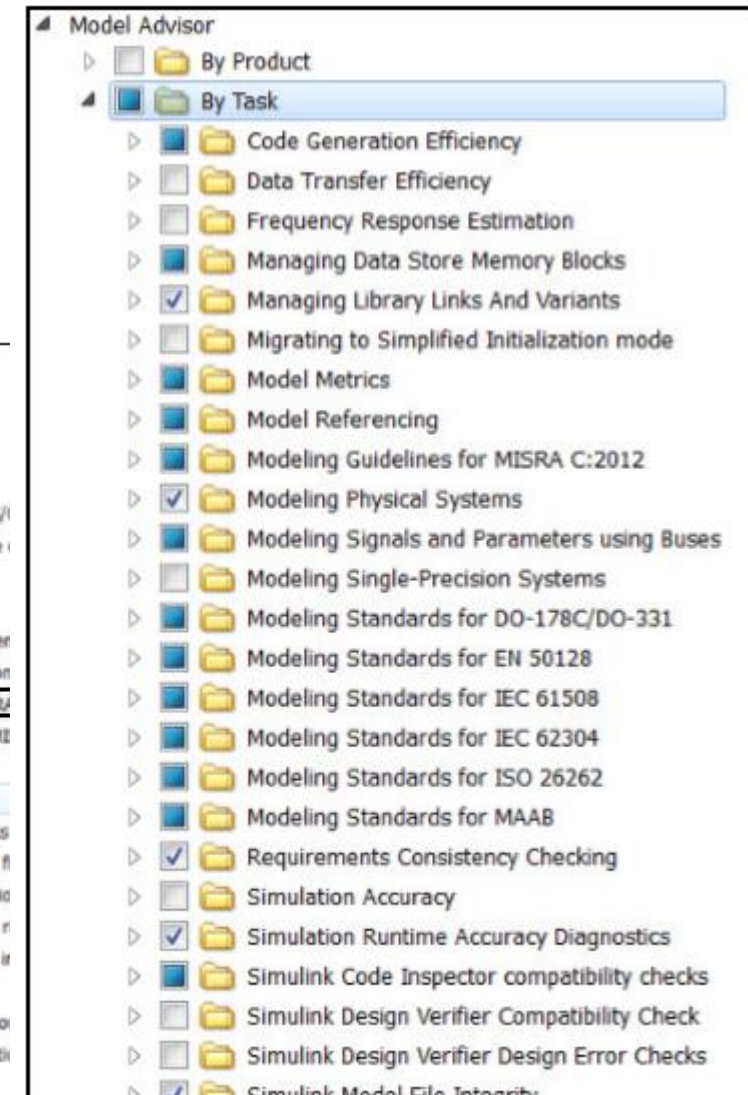
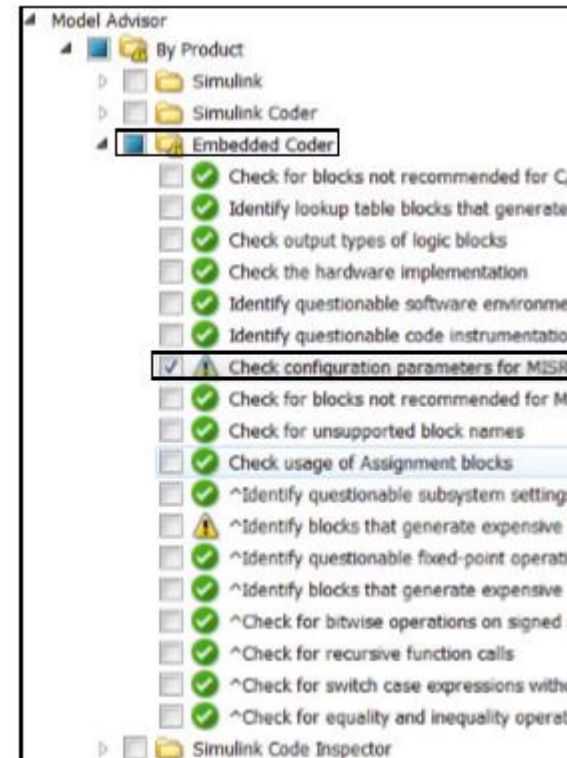
# Compliance with standards and guidelines

If your application has mission-critical development and certification goals, your models or subsystems and the code generated for them might need to comply with one or more of the standards and guidelines listed below.

- Standard: DO-178C, Software Considerations in Airborne Systems and Equipment Certification. The DO-331 standard is a companion document for modeling standards.
- Standard: IEC 61508, Functional safety of electrical/ electronic/ programmable electronic safety-related systems.
- Standard: ISO 26262, an international functional safety standard titled Road vehicles.
- Standard: AUTomotive Open System ARchitecture (AUTOSAR).
- Guidelines: Use of the C Language in Critical Systems (MISRA C®)
- Guidelines: MAAB, use of MATLAB, Simulink, and Stateflow® software for control algorithm modeling.

The Model Advisor provides built-in checks to verify model and code compliance with DO-178C, ISO 26262 and IEC 61508 standards as well as MAAB guidelines. You can use one of the Model Advisor checks for Embedded Coder to verify the compliance of your code with MISRA C guidelines.

Note The Model Advisor checks for standards and guidelines are available only if you have a license for Simulink Check™ product.



## Focused Product :

[Simulink](#)

[Simulink Coder](#)

[Embedded Coder](#)

# Q&A

## Support:

[erkam.cankaya@figes.com.tr](mailto:erkam.cankaya@figes.com.tr)  
[esra.bozkurt@figes.com.tr](mailto:esra.bozkurt@figes.com.tr)

[matlab.support@figes.com.tr](mailto:matlab.support@figes.com.tr)