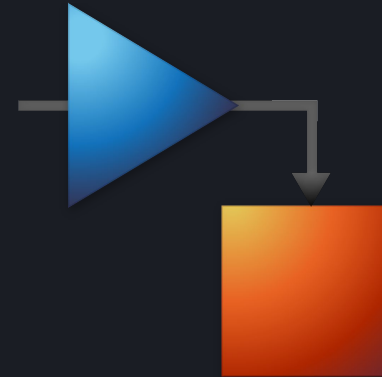


Subsystem, Solvers and code Generation Process

Session Content

- Commonly Used Blocks
- Code Generation Process
- Subsystems

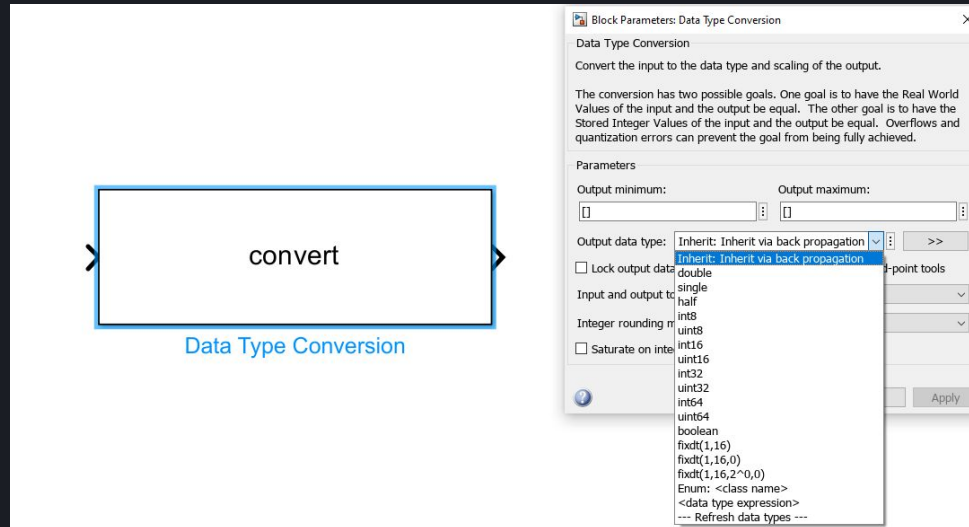


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Data Type Conversion

In Simulink, the Data Type Conversion block is used to convert signals from one data type to another. This block is particularly useful when working with systems that require different data types for compatibility or precision reasons. It allows you to seamlessly convert signals between various numeric data types, such as changing from a double-precision floating-point to a single-precision floating-point or from an integer to a floating-point format.



Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Data Type Conversion
 - Double to Single Precision Conversion:
 - Input data type: double
 - Output data type: single

This configuration of the Data Type Conversion block converts a double-precision input signal to a single-precision output signal. This may be useful when working with systems that require lower memory usage or when interfacing with components that only accept single-precision data.

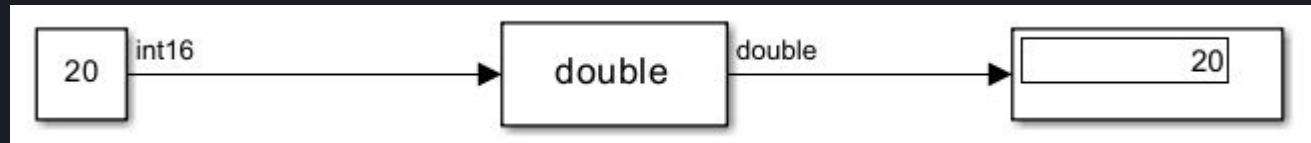


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Data Type Conversion
 - Integer to Floating-Point Conversion:
 - Input data type: int16
 - Output data type: double

In this example, the Data Type Conversion block is used to convert a 16-bit integer input signal to a double-precision floating-point output signal. This type of conversion is common when performing calculations that require higher precision.

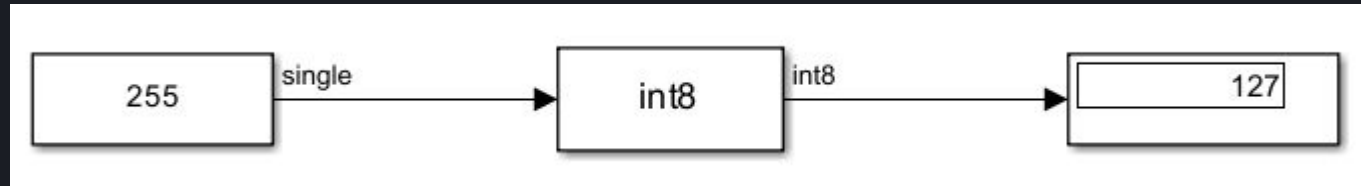


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Data Type Conversion
 - Rounding and Overflow Handling:
 - Input data type: single
 - Output data type: int8
 - Rounding mode: Round
 - Overflow mode: Saturate

This example converts a single-precision floating-point input signal to an 8-bit integer output signal with rounding and overflow handling. The rounding mode is set to "Round," and overflow handling is set to "Saturate," which means that values exceeding the range of the output data type will be saturated to the minimum or maximum representable value.

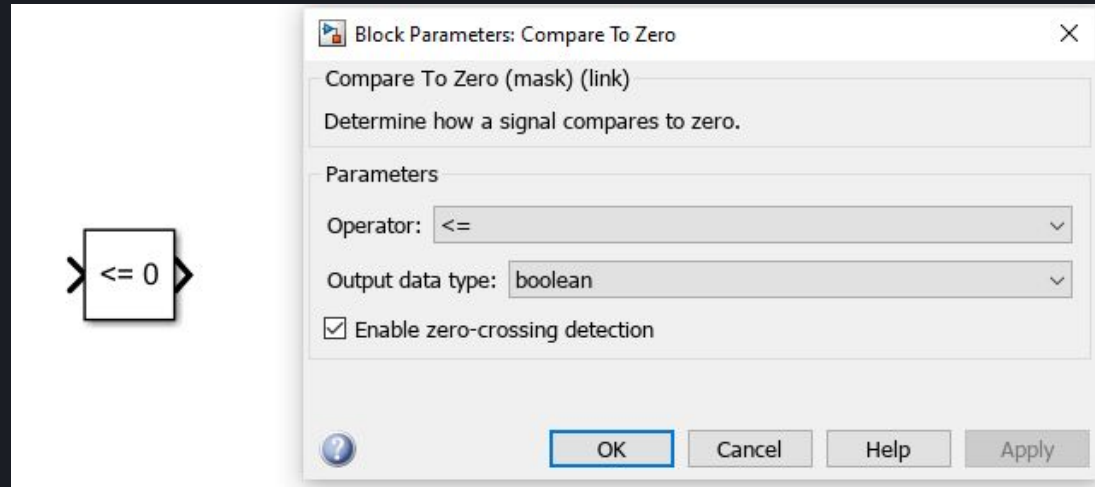


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Compare To Zero

In Simulink, the "Compare to Zero" block is similar to the "Compare to Constant" block, but it specifically compares an input signal to zero. It outputs a logical (Boolean) value indicating whether the input signal is greater than, less than, equal to, or not equal to zero. This block is often used for conditional checks and logical operations where the comparison is with zero.

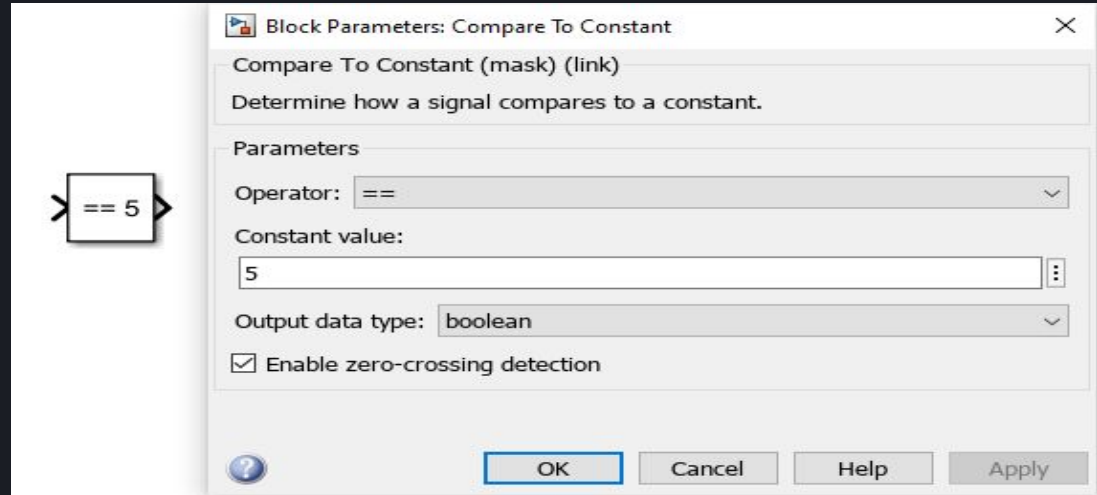


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Compare To Constant

In Simulink, the "Compare to Constant" block is used to compare an input signal to a specified constant value. It outputs a logical (Boolean) value indicating whether the input signal is greater than, less than, equal to, or not equal to the specified constant. This block is commonly used in control systems, logical decision-making, and conditional operations.

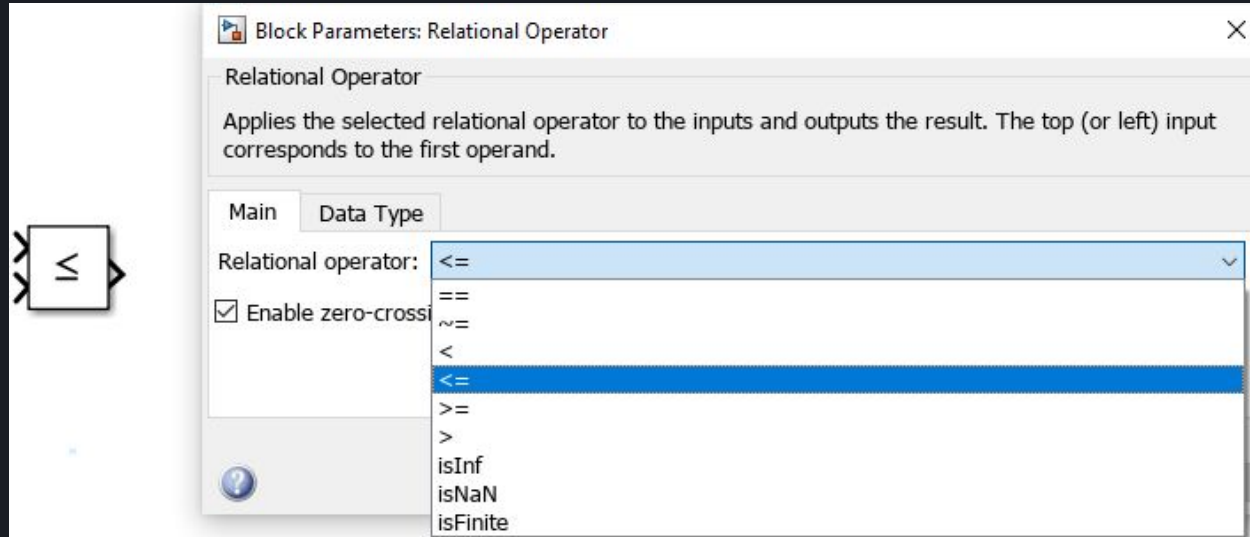


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Relational Operator block

In Simulink, relational operators are used to perform comparisons between signals or operands. These operators evaluate conditions and produce logical (Boolean) outcomes, such as true or false. Common relation operators include

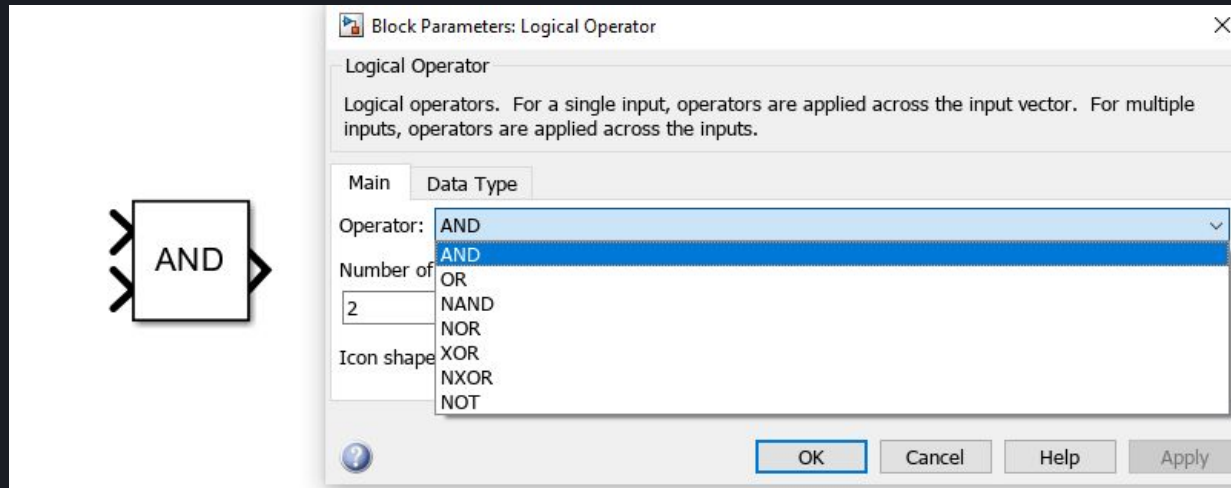


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Logical Operator block

In Simulink, logical operators are used to perform logical operations on signals. These operators evaluate conditions and produce logical (Boolean) outcomes, such as true or false. Common logical operators include AND, OR, NOT, XOR, and others.

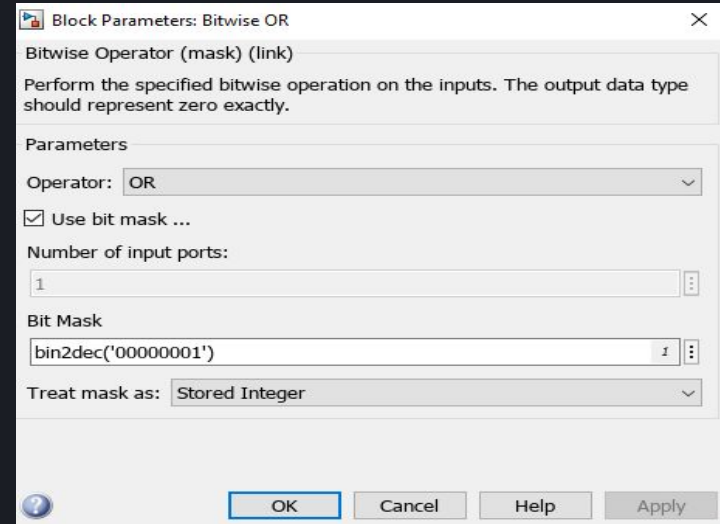
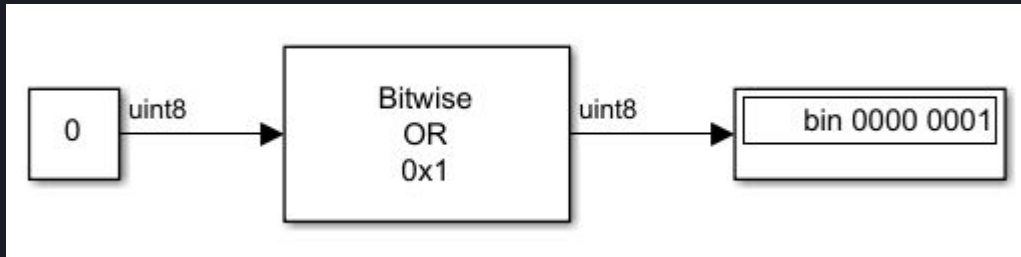


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Bitwise Operator block

In Simulink, bitwise operators are used to perform bitwise operations on integer signals. Unlike logical operators that operate on Boolean (true/false) values, bitwise operators work at the level of individual bits within integer signals. Common bitwise operators include AND, OR, XOR, NOT, and bit-shifting operators.

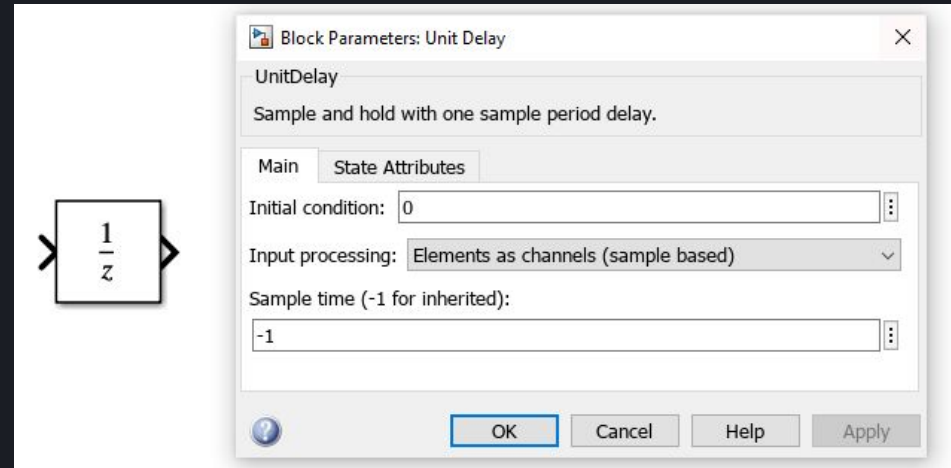


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Unit Delay

In Simulink, the "Unit Delay" block represents a one-sample delay in a discrete-time system. It is commonly used to model systems where the output at the current time step depends on the input at the previous time step. The unit delay block helps capture the dynamic behavior of systems and is fundamental for building discrete-time models.

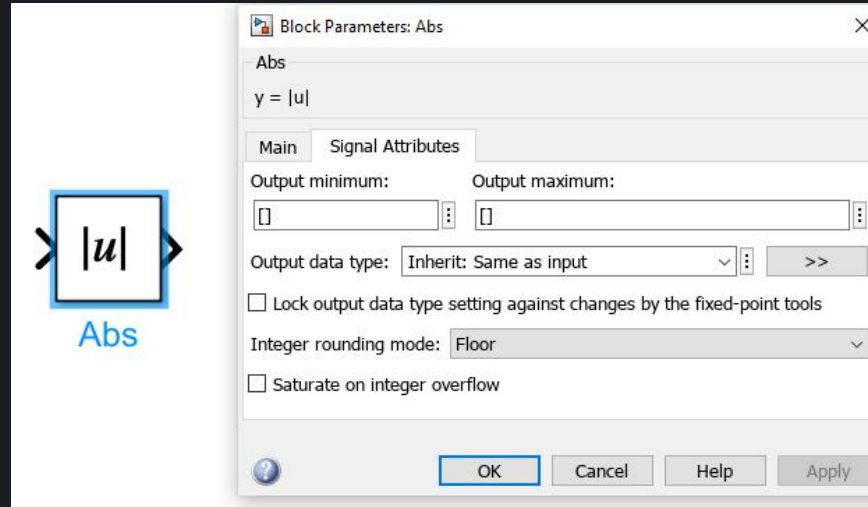


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Abs block

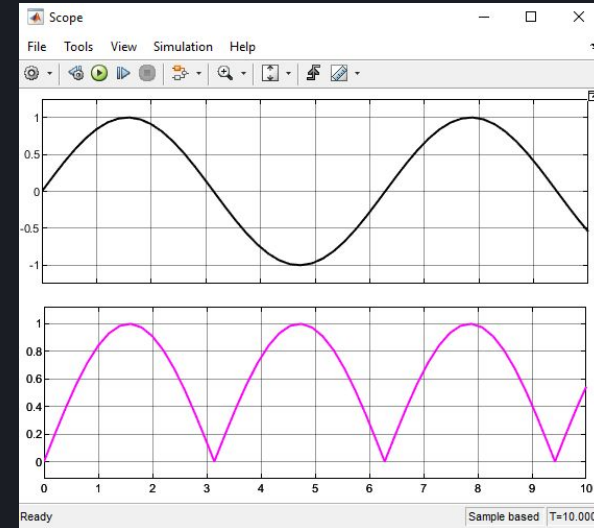
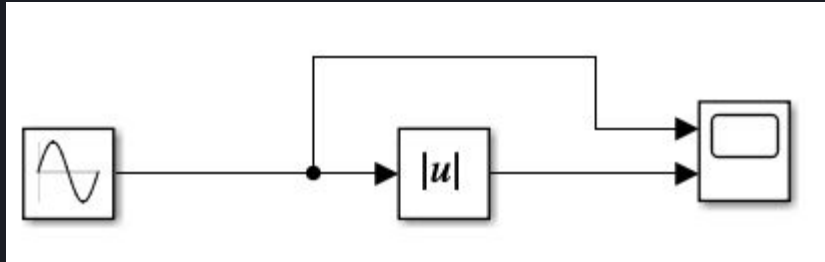
In Simulink, the "Abs" block is used to compute the absolute value of an input signal. The absolute value of a number is its distance from zero on the number line, regardless of the direction. The "Abs" block outputs the magnitude of the input signal.



Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Abs block

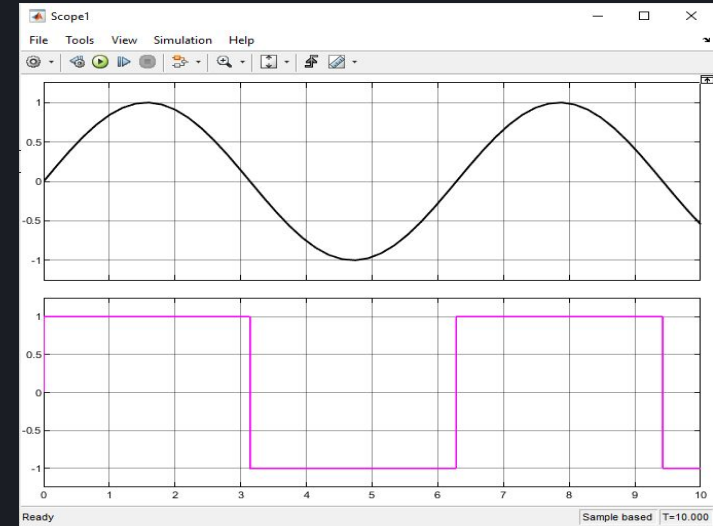
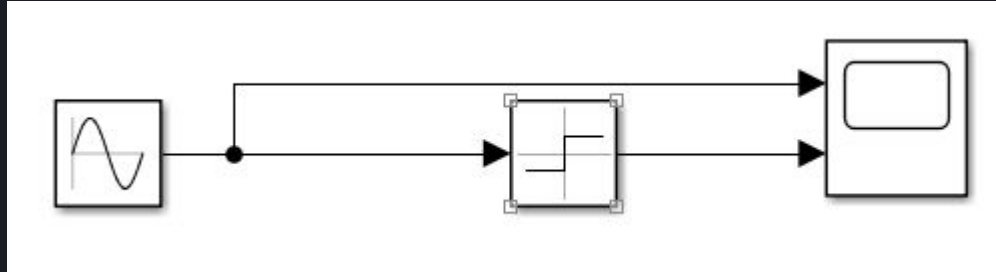


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- sign block

In Simulink, the "Sign" block is used to determine the sign of an input signal. It outputs a signal that represents the sign of the input, which can be either positive, negative, or zero. The output typically takes the values +1 for positive input, -1 for negative input, and 0 for zero input.

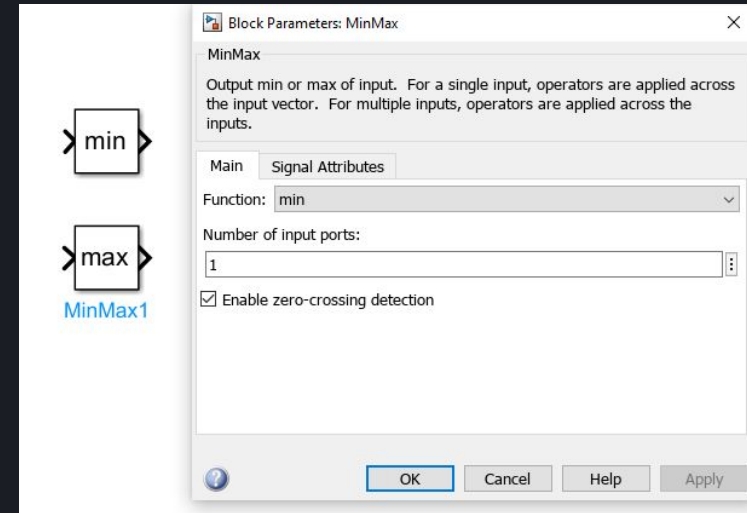


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- MinMax block

In Simulink, the "MinMax" block is used to compute the minimum and maximum values of two input signals. This block helps determine the minimum and maximum values between two signals at each simulation time step.

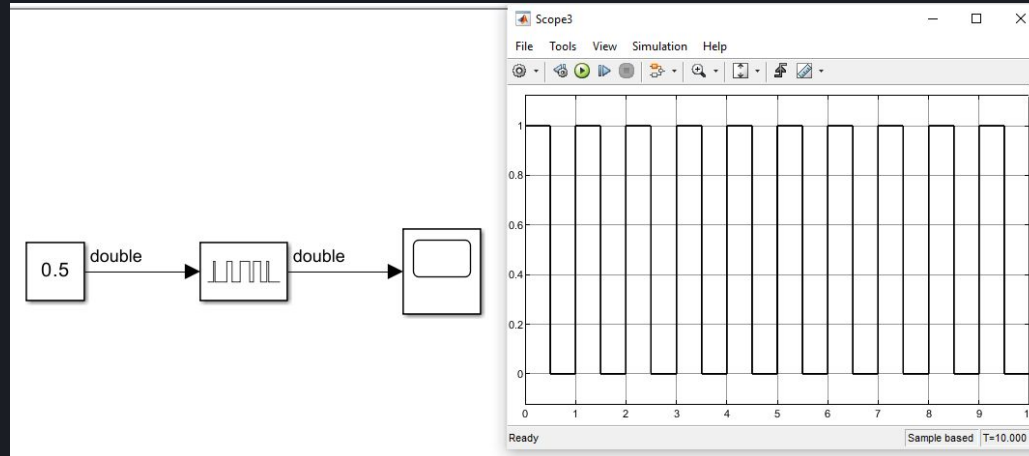


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- PWM block

In Simulink, the "PWM (Pulse Width Modulation)" block is used to generate a pulse-width modulated signal. Pulse Width Modulation is a technique commonly employed in control systems and power electronics to control the average value of a signal by varying the width of its pulses. PWM is often used in applications such as motor control, power inverters, and communication systems.



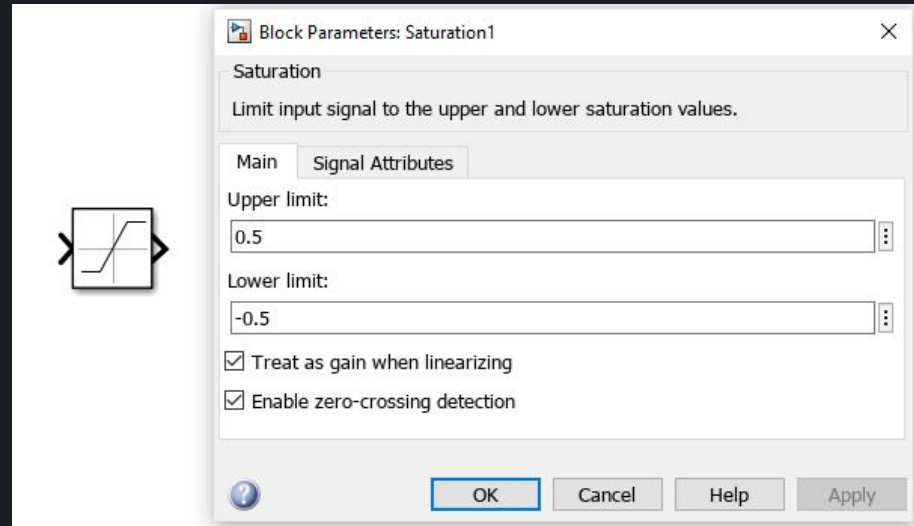
Model-Based Development Program

Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Saturation block

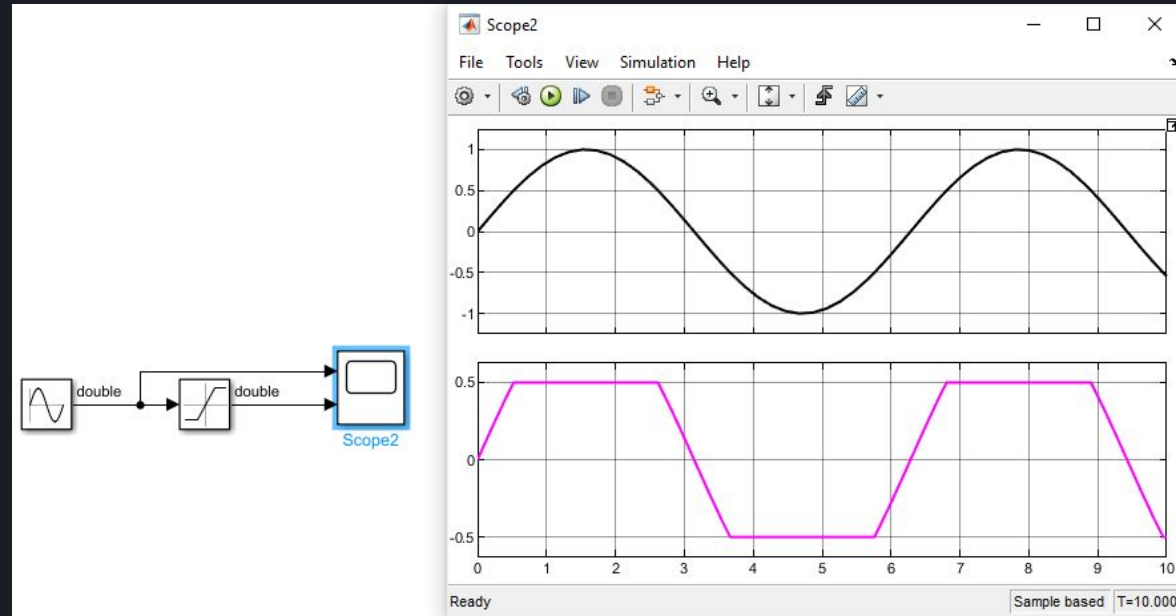
In Simulink, the "Saturation" block is used to limit the range of an input signal within a specified upper and lower bound. This block is particularly useful for preventing signals from exceeding certain limits, providing a way to model physical constraints or constraints imposed by a system.



Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Saturation block

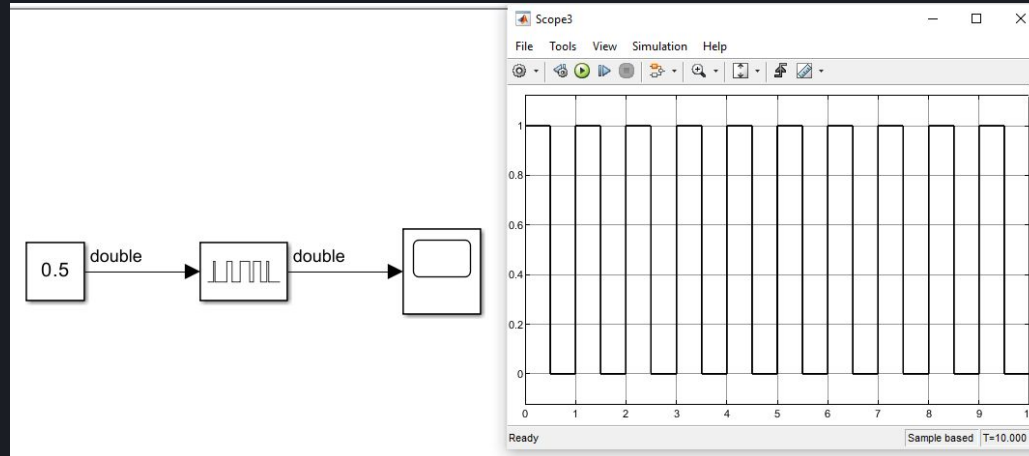


Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- PWM block

In Simulink, the "PWM (Pulse Width Modulation)" block is used to generate a pulse-width modulated signal. Pulse Width Modulation is a technique commonly employed in control systems and power electronics to control the average value of a signal by varying the width of its pulses. PWM is often used in applications such as motor control, power inverters, and communication systems.



Model-Based Development Program

Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Goto and from block
In Simulink, the "Goto" and "From" blocks are used to create signal connections between different parts of a model. These blocks help organize and simplify the layout of a Simulink model by allowing you to create explicit signal connections without drawing lines across the entire diagram.



Subsystem, Solvers and code Generation Process

Commonly Used Blocks

- Saturation block

**Now, Try to implement Saturation Block
With another methods**

Subsystem, Solvers and code Generation Process

[Quiz 3: Click Here To Start](#)

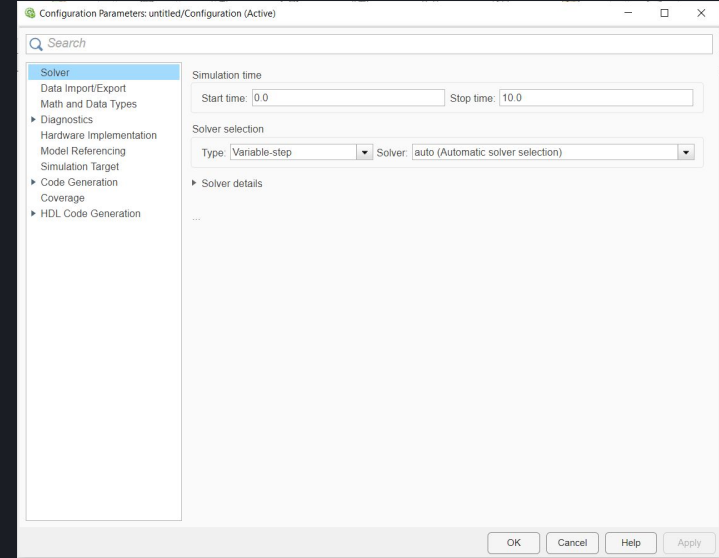


Subsystem, Solvers and code Generation Process

Solvers (How Simulink Simulate Models?)

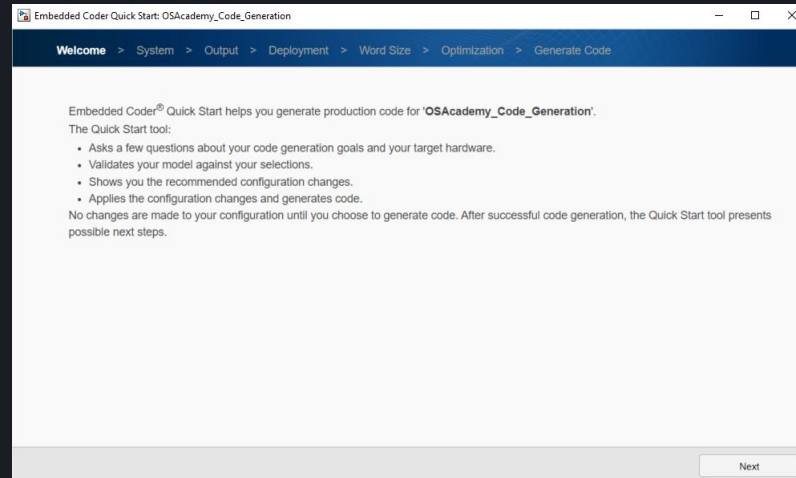
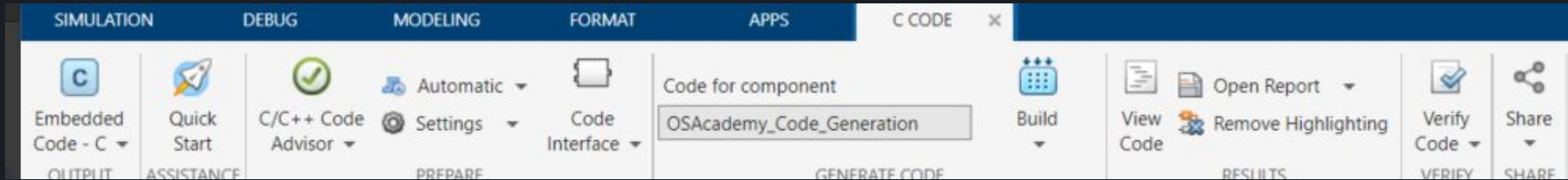
In Simulink, solvers are algorithms used to numerically solve the ordinary differential equations (ODEs) that describe the behavior of dynamic systems. These solvers determine how the simulation advances in time and how accurately it captures the system's behavior. Simulink provides various solver options to address different simulation requirements. Here's an explanation of solvers in Simulink:

1. **Fixed-Step Solvers:** These solvers use a constant time step during the simulation. They are suitable for systems with a known and constant step size. Common fixed-step solvers include the Fixed Step and Variable Step solvers.
2. **Variable-Step Solvers:** Variable-step solvers adjust the time step dynamically based on the system's behavior. They are more efficient for systems with varying dynamics or when a high level of accuracy is needed. Common variable-step solvers include ODE1, ODE2, and ODE3.



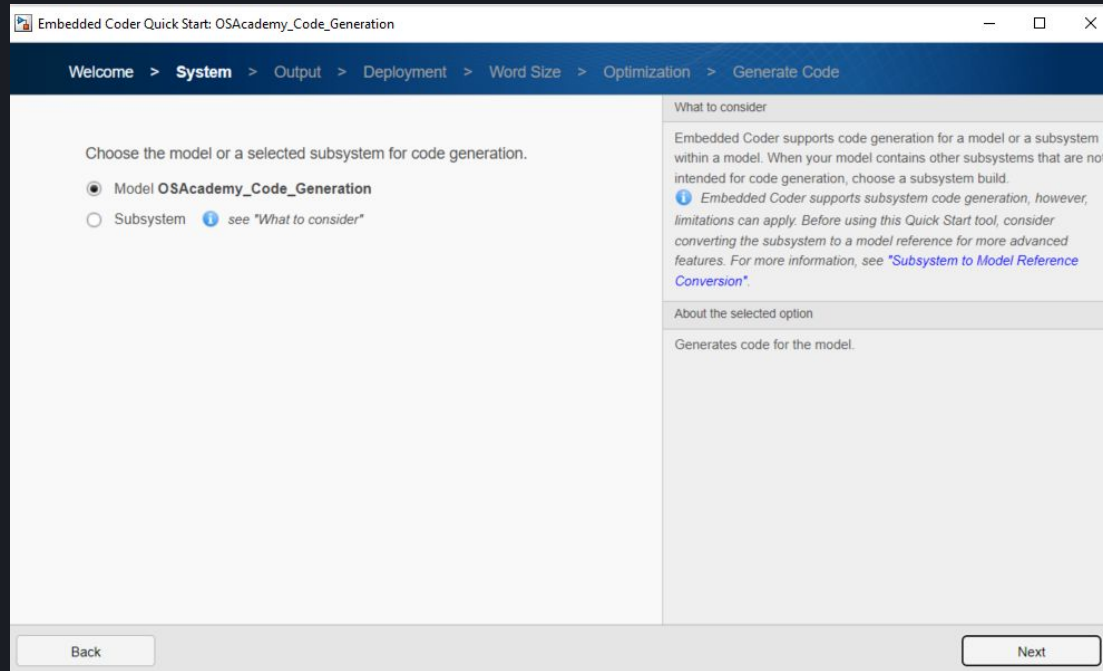
Subsystem, Solvers and code Generation Process

Code Generation Process



Subsystem, Solvers and code Generation Process

Code Generation Process



Subsystem, Solvers and code Generation Process

Code Generation Process

The screenshot shows a software window titled "Embedded Coder Quick Start: OSAcademy_Code_Generation". The window has a blue header bar with a breadcrumb trail: "Welcome > System > **Output** > Deployment > Word Size > Optimization > Generate Code". The main content area is divided into two columns. The left column contains two sections: "Select the output for your generated code." with four radio button options (C code is selected), and "How many instances do you need for your generated code?" with two radio button options (Single instance is selected). The right column contains two informational sections: "What to consider" and "About the selected option". At the bottom of the window are "Back" and "Next" buttons.

Embedded Coder Quick Start: OSAcademy_Code_Generation

Welcome > System > **Output** > Deployment > Word Size > Optimization > Generate Code

Select the output for your generated code.

- ☒ C code
- ☐ C code compliant with AUTOSAR
- ☐ C++ code
- ☐ C++ code compliant with AUTOSAR Adaptive Platform

How many instances do you need for your generated code?

- ☒ Single instance
- ☐ Multiple instances

What to consider

Embedded Coder® supports several different types of code output. Select the output that best fits your application.

For C and AUTOSAR C code, you can choose to configure your model for multi-instance code generation. Select multi-instance if your application can benefit from reentrant code and requires that each use or instance of the code maintains its own unique data.

About the selected option

Embedded Coder generates code that is compliant with ANSI/ISO C.

For successful multi-instance code generation, your model might require changes beyond the scope of the Quick Start configuration.

Back Next

Subsystem, Solvers and code Generation Process

Code Generation Process

The image displays two sequential screenshots of the 'Embedded Coder Quick Start: OSAcademy_Code_Generation' application window. The left screenshot shows the 'Deployment' step in the navigation pane, with a message: 'Analyzing your model to determine how to deploy your generated code. This analysis can take some time.' Below this, a box states: 'The Quick Start tool examines your model to determine:' followed by a list of four questions: 'How many sample rates are in your system?', 'Does your system contain continuous states?', 'Did you configure your system for export function calls?', and 'Does your system contain referenced models?'. The right screenshot shows the 'Generate Code' step. It displays 'Analysis is complete.' followed by a table of analysis results for 'OSAcademy_Code_Generation'. The table has two columns: the question and the answer. The results are: 'Single rate' for sample rates, 'No' for continuous states, 'No' for export function calls, and 'No' for referenced models. A blue link 'Read more about analysis' is present. Below the table, a warning message states: 'Your model is configured with a variable-step solver. Production code generation does not support variable-step solvers. If you choose to generate code, the Quick Start tool changes the solver to fixed-step. Changing solvers can change your simulation results. For information on choosing a solver, see here.' Both screenshots have a 'Back' button at the bottom left. The right screenshot also has a 'Next' button at the bottom right.

Embedded Coder Quick Start: OSAcademy_Code_Generation

Welcome > System > Output > **Deployment** > Word Size > Optimization

Analyzing your model to determine how to deploy your generated code.
This analysis can take some time.

The Quick Start tool examines your model to determine:

- How many sample rates are in your system?
- Does your system contain continuous states?
- Did you configure your system for export function calls?
- Does your system contain referenced models?

Back

Embedded Coder Quick Start: OSAcademy_Code_Generation

Welcome > System > Output > **Deployment** > Word Size > Optimization > **Generate Code**

Analysis is complete.

Analysis results for: 'OSAcademy_Code_Generation' [Read more about analysis](#)

How many sample rates are in your system?	Single rate
Does your system contain continuous states?	No
Did you configure your system for export function calls?	No
Does your system contain referenced models?	No

i Your model is configured with a variable-step solver. Production code generation does not support variable-step solvers. If you choose to generate code, the Quick Start tool changes the solver to fixed-step. Changing solvers can change your simulation results. For information on choosing a solver, see [here](#).

Back

Next

Subsystem, Solvers and code Generation Process

Code Generation Process

Embedded Coder Quick Start: OSAcademy_Code_Generation

Welcome > System > Output > Deployment > **Word Size** > Optimization > Generate Code

Select your target hardware processor type. If your hardware processor is not listed, select "Custom Processor" to define your data type sizes.

Device Vendor:

Device Type:

Number of bits

char:	<input type="text" value="8"/>	short:	<input type="text" value="16"/>	int:	<input type="text" value="16"/>
long:	<input type="text" value="32"/>	long long:	<input type="text" value="64"/>	native:	<input type="text" value="8"/>
pointer:	<input type="text" value="16"/>	size_t:	<input type="text" value="16"/>	ptrdiff_t:	<input type="text" value="16"/>

What to consider

Simulink and Embedded Coder use target hardware information to determine integer math behavior and to produce bit-true results between simulation and generated code. Embedded Coder leverages this information to perform target-specific optimizations.

Back Next

Intel

AMD

ARM Compatible

Altera

Analog Devices

Apple

Atmel

Freemscale

Infineon

Intel

Microchip

NXP

RISC-V

Renesas

STMicroelectronics

Texas Instruments

Custom Processor

Subsystem, Solvers and code Generation Process

Code Generation Process

The screenshot shows a software window titled "Embedded Coder Quick Start: OSAcademy_Code_Generation". The window has a blue header bar with a breadcrumb trail: "Welcome > System > Output > Deployment > Word Size > **Optimization** > Generate Code".

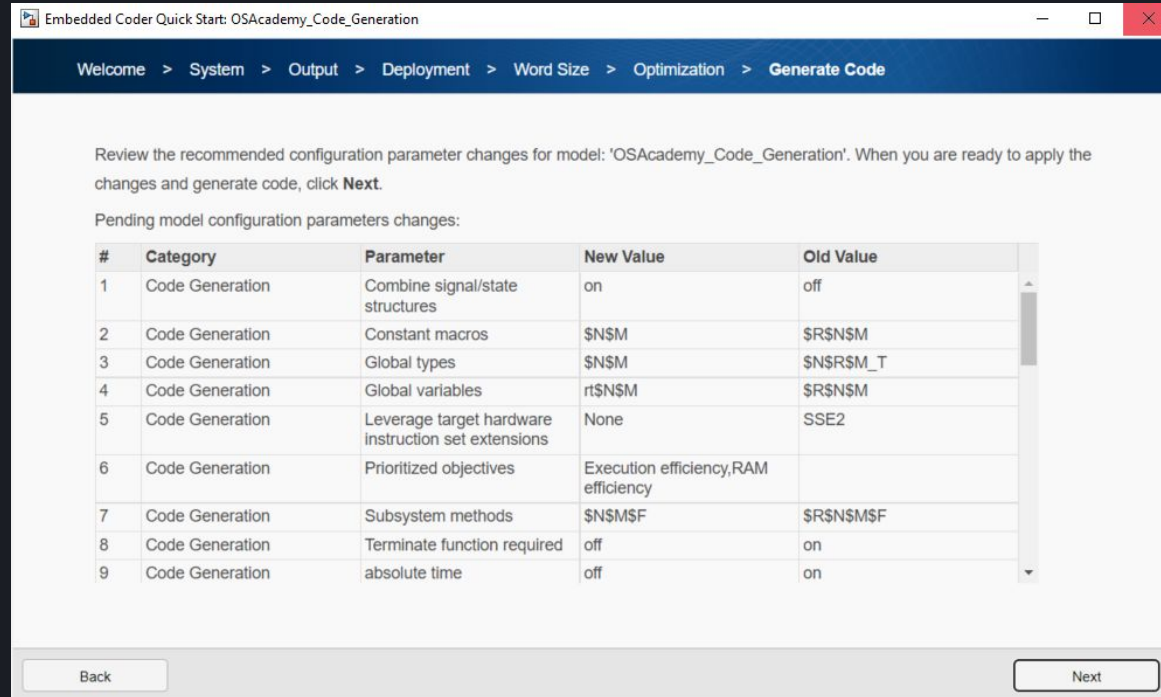
The main content area is divided into two panels. The left panel contains the text "Select your most important code generation objective." followed by two radio button options: "Execution efficiency" (which is selected) and "RAM efficiency".

The right panel contains two sections of text. The first section, titled "What to consider", states: "Based on your selection, the Quick Start tool configures your model with the best optimizations for your specified code generation objective." followed by a tip: "After Quick Start code generation is complete, you can fine-tune your optimization settings using the Code Generation Advisor." The second section, titled "About the selected option", states: "If faster execution is your primary objective, select this option. The Quick Start tool configures the model to optimize code for execution speed and RAM usage. The priority is the execution speed."

At the bottom of the window, there are two buttons: "Back" on the left and "Next" on the right.

Subsystem, Solvers and code Generation Process

Code Generation Process



Subsystem, Solvers and code Generation Process

Code Generation Process

Code Generation Complete

Quick Start code generation is complete.

- [View code generation report](#)

What's Next

Now that Quick Start has configured your model for basic code generation, the following links present possible next steps:

- [Preserve variables in generated code](#)
- [Fine-tune optimizations using the Code Generation Advisor](#)
- [Customize the appearance of your generated code](#)
- [Build your generated code](#)
- [Manage your model configuration](#)
- [Regenerate code](#)
- [Package code for sharing](#)

[Revert Configuration](#)

Code Generation Report

Find: Match Case

OSAcademy_Code_Generation ▼

Content

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

Code

- ▼ Main file
 - ert_main.c
- ▼ Model files
 - OSAcademy_Code_Gen
 - OSAcademy_Code_Gen
- ▼ Utility files
 - rtwtypes.h

Code Generation Report for 'OSAcademy_Code_Generation'

Model Information

Author	Abdelrhman Shaban
Last Modified By	Abdelrhman Shaban
Model Version	1.2
Tasking Mode	SingleTasking

[Configuration settings at time of code generation](#)

Code Information

System	ert.tlc
Target File	
Hardware Device	Atmel->AVR
Type	
Simulink Coder Version	9.8 (R2022b) 13-May-2022
Timestamp of Generated	Wed Oct 25 17:02:35 2023

[OK](#) [Help](#)

Subsystem, Solvers and code Generation Process

Code Generation Process

Code

▼ Main file

ert_main.c

▼ Model files

OSAcademy_Code_Generation.c

OSAcademy_Code_Generation.h

▼ Utility files

rtwtypes.h

Using Embedded Coder in Simulink streamlines the process of generating code for embedded systems, making it easier to transition from simulation to real-world deployment. This is a critical step in the development of embedded control systems and helps ensure the reliability and performance of your systems in real-time applications.

Subsystem, Solvers and code Generation Process

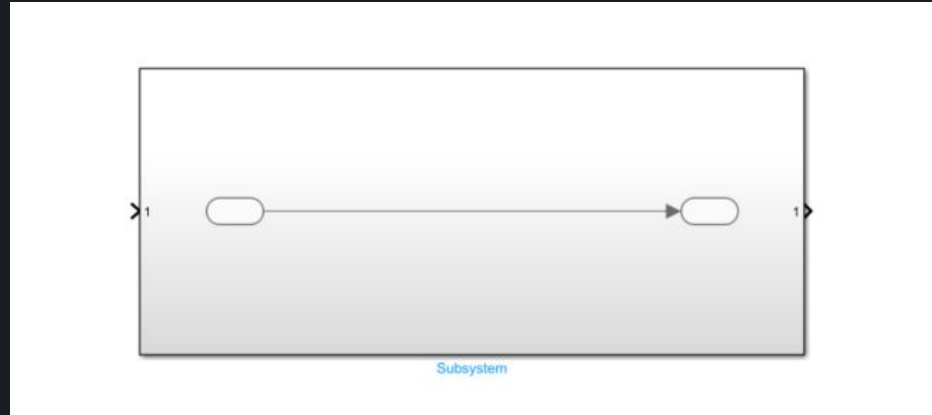
[Quiz 4: Click Here To Start](#)



Subsystem, Solvers and code Generation Process

Subsystems

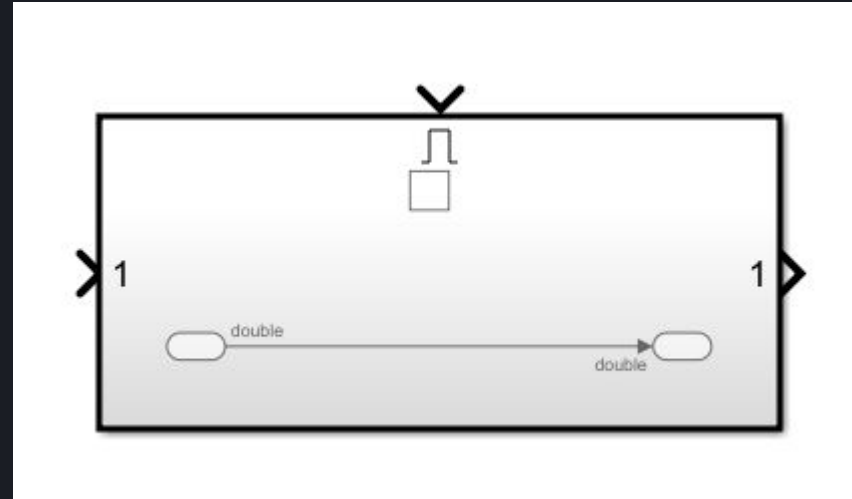
- Subsystem



Subsystem, Solvers and code Generation Process

Subsystems

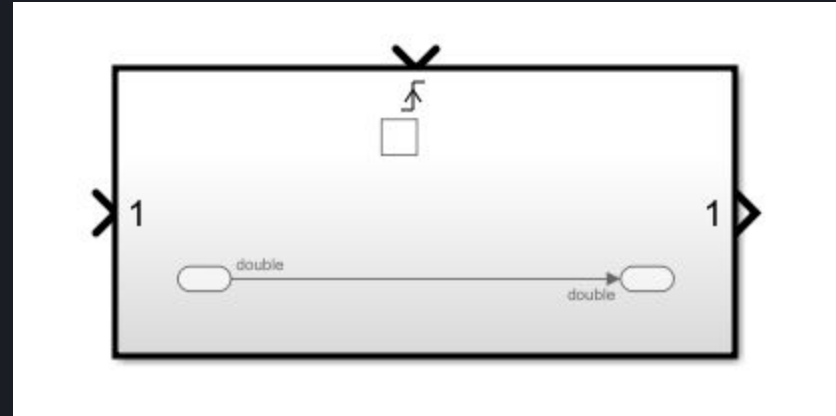
- Enable Subsystem



Subsystem, Solvers and code Generation Process

Subsystems

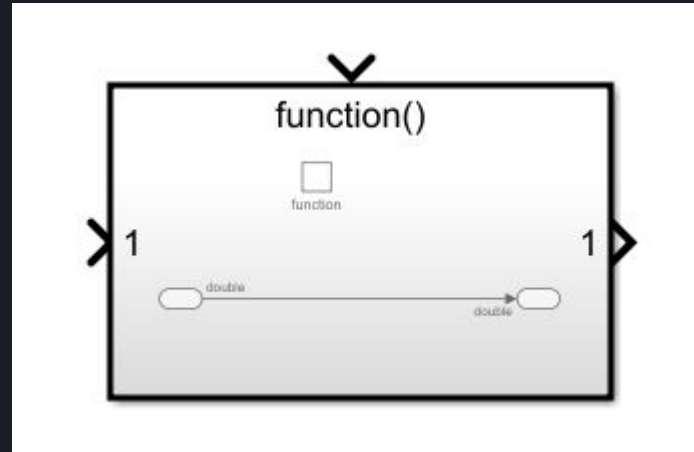
- Triggered Subsystem



Subsystem, Solvers and code Generation Process

Subsystems

- Function Call Subsystem



Subsystem, Solvers and code Generation Process



[Lab 2: Click Here To Start](#)