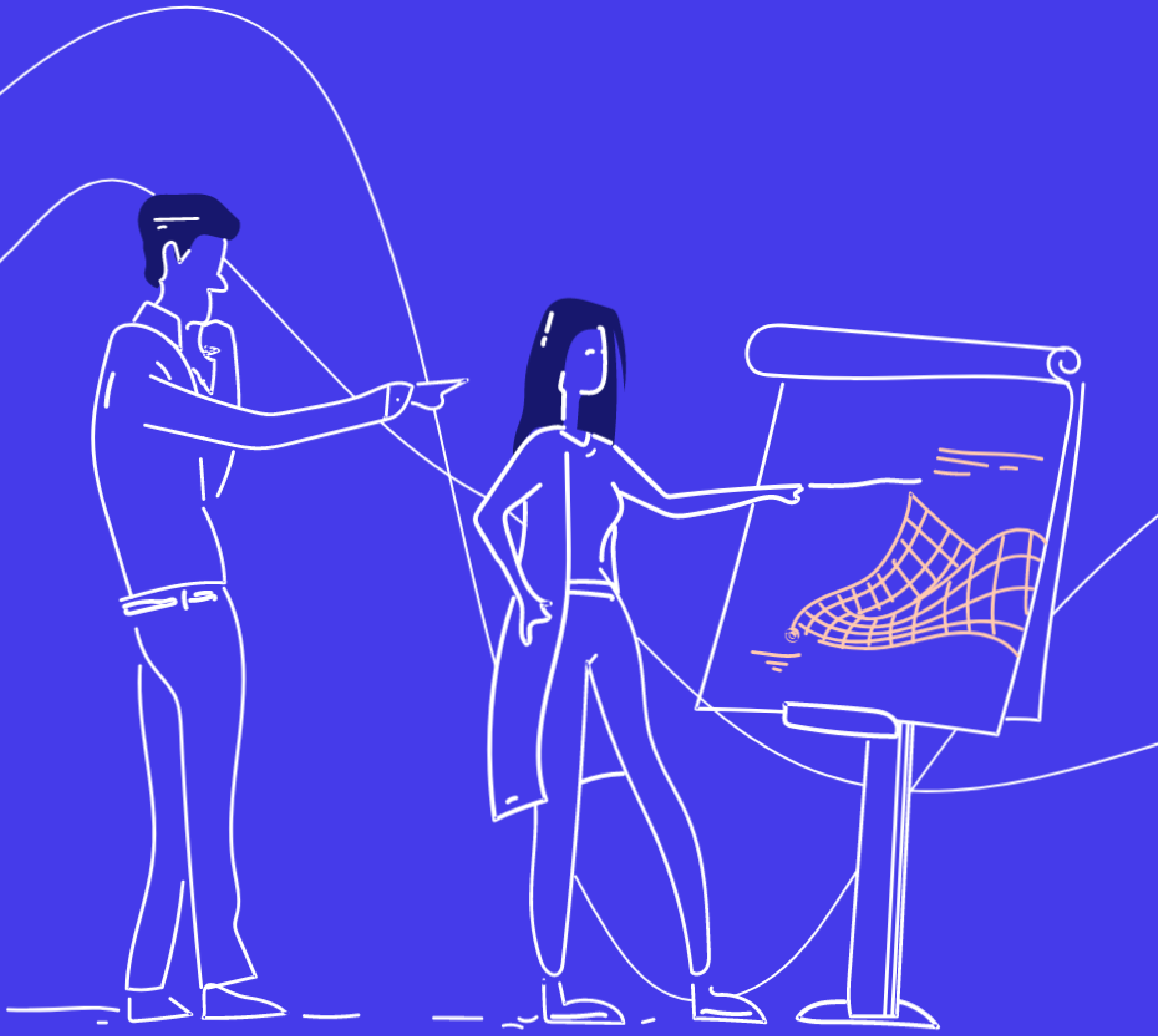
sciengineer

Embedded Coder for Production Code Generation



SciEngineer's training courses are designed to help organizations and individuals close skills gaps, keep up-to-date with the industry-accepted best practices and achieve the greatest value from MathWorks® and COMSOL® Products.

Embedded Coder for Production Code Generation

Embedded Coder for Production Code Generation

This three-day course describes techniques for generating, validating, and customizing embedded code using Embedded Coder.

Topics include: Generated code structure and execution; Code generation options and optimalizations; Integrating generated code with external code; Generating code for multirate systems; Customizing generated code and data.

Prerequisites

- Simulink Fundamentals (or Simulink Fundamentals for Automotive Applications or Simulink Fundamentals for Aerospace Applications)
- Knowledge of C programming language.

DURATION	LEVEL
3 Days	Medium
●●●	▬▬▭

TOPICS

Day 1

- Generating Embedded Code
- Optimizing Generated Code
- Integrating Generated Code with External Code
- Controlling Function Prototypes

Day 2

- Customizing Data Characteristics in Simulink®
- Customizing Data Characteristics Using Data Objects
- Creating Storage Classes
- Customizing Generated Code Architecture
- Model Referencing and Bus Objects

Day 3

- Scheduling Generated Code Execution
- Testing Generated Code on Target Hardware
- Deploying Generated Code
- Integrating Device Drivers
- Improving Code Efficiency and Compliance

SciEngineer's Training Services

Generating Embedded Code

OBJECTIVE: Configure Simulink models for embedded code generation and effectively interpret the generated code.

- Architecture of an embedded application
- System specification
- Generating code
- Code modules
- Logging intermediate signals
- Data structures in generated code
- Verifying generated code
- Embedded Coder® build process

Optimizing Generated Code

OBJECTIVE: Identify the requirements of the application at hand and configure optimization settings to satisfy these requirements.

- Optimization considerations
- Removing unnecessary code
- Removing unnecessary data support
- Optimizing data storage
- Profiling generated code
- Code generation objectives

Integrating Generated Code with External Code

OBJECTIVE: Modify models and files to run generated code and external code together.

- External code integration overview
- Model entry points
- Creating an execution harness
- Integrating generated code into an external project
- Controlling code destination
- Packaging generated code

Controlling Function Prototypes

OBJECTIVE: Customize function prototypes of model entry points in the generated code.

- Default model function prototype
- Modifying function prototypes
- Generated code with modified function prototypes
- Model function prototype considerations
- Reusable function interface
- Function defaults

Customizing Data Characteristics Using Data Objects

OBJECTIVE: Control the data types and storage classes of data using data objects.

- Simulink® data objects overview
- Controlling data types with data objects
- Creating reconfigurable data types
- Controlling storage classes with data objects
- Controlling data type and variable names
- Data dictionaries

Customizing Generated Code Architecture

OBJECTIVE: Control the architecture of the generated code according to application requirements.

- Simulink model architecture
- Controlling code partitioning
- Generating reusable subsystem code
- Generating variant components
- Code placement option

Model Referencing and Bus Objects

OBJECTIVE: Control the data type and storage class of bus objects and use them for generating code from models that reference other models.

- Creating reusable model references
- Controlling data type of bus signals
- Controlling storage class of bus signals
- Model Reference software testing

Customizing Data Characteristics in Simulink®

OBJECTIVE: Control the data types and storage classes of data in Simulink.

- Data characteristics
- Data type classification
- Simulink data type configuration
- Setting signal storage classes
- Setting state storage classes
- Impact of storage classes on symbols

Creating Storage Classes

OBJECTIVE: Design storage classes and use them for code generation.

- User-defined storage classes
- Creating storage classes
- Using user-defined storage classes
- Sharing user code definitions

Scheduling Generated Code Execution

OBJECTIVE: Generate code for multi-rate systems in single-tasking, multitasking, and function call-driven configurations.

- Execution schemes for single-rate and multi-rate systems
- Generated code for single-rate models
- Multi-rate single-tasking code
- Multi-rate multitasking code
- Generating exported functions

Improving Code Efficiency and Compliance

OBJECTIVE: Inspect the efficiency of generated code and verify compliance with standards and guidelines.

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

Testing Generated Code on Target Hardware

OBJECTIVE: Use processor-in-the-loop (PIL) simulation to validate, profile, and optimize the generated code on target hardware.

- Hardware support overview
- Arduino setup
- Validating generated code on target
- Target optimization overview
- Profiling generated code on target
- Using code replacement libraries
- Creating code replacement tables

Deploying Generated Code

OBJECTIVE: Create a working real-time application on an Arduino® board using provided hardware support.

- Embedded application architecture
- Creating a deployment harness
- Using device driver blocks
- Running a real-time application
- External mode

Integrating Device Drivers

OBJECTIVE: Generate custom blocks to integrate device drivers with Simulink and generated code.

- Device drivers overview
- Using Legacy Code Tool
- Customizing device driver components
- Developing a device driver block for Arduino

sciengineer

Expand your knowledge