The use of Model-Based Design in the automotive market has expanded throughout the last few decades. One main use case has been to use MATLAB® and Simulink® to develop and deploy algorithms for automotive embedded applications. Engineers have used Model-Based Design to develop applications such as engine controllers, transmission controllers, body controllers, and more recently, autonomous driving and advanced driver-assistance systems. As automotive embedded applications evolve to rely less on driver input, the embedded systems that control the vehicle will increasingly need to meet functional safety requirements.

To increase the likelihood that functional safety requirements for these systems are met, standards have been developed to guide engineers through various aspects of the development cycle. One standard that has gained traction in the automotive functional safety space is ISO 26262. ISO 26262 uses a top-down workflow and breaks down various aspects of the development cycle including system-, hardware-, and software-level guidelines to achieve functional safety objectives.

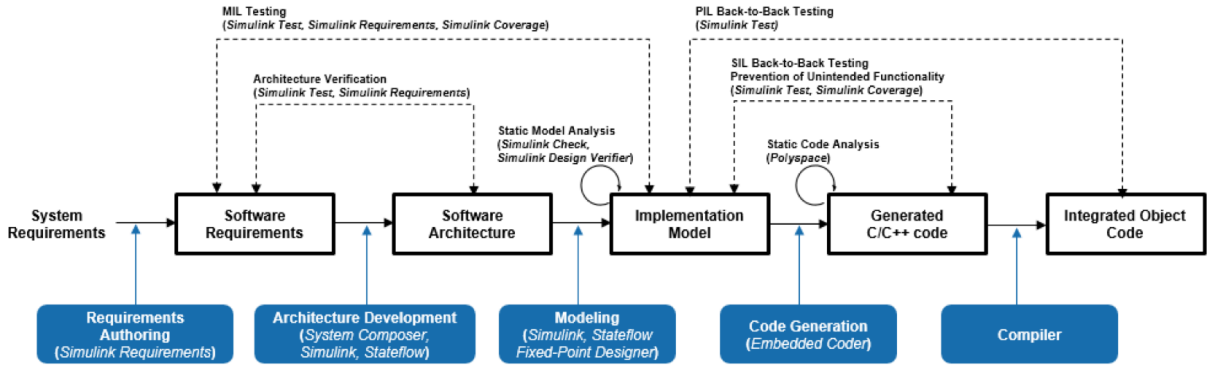
IEC Certification Kit can be used as a guide to compliance with the portions of the ISO 26262 standard that are applicable to Model-Based Design. The kit provides a high-level workflow that you can reference and extend as needed. One thing that you will need to carefully consider is the modeling style and architecture that you will use when constructing algorithms. By adopting a modeling style and architecture that are conducive to ISO 26262, you can greatly reduce the work needed to meet key aspects of the standard such as freedom from interference. This white paper details a variety of modeling best practices that you can use to form a modeling architecture that complements the referenced workflow in IEC Certification Kit. MathWorks Consulting Services has uncovered these best practices during consulting engagements with various automotive companies.

Why ISO 26262?

The ISO 26262 standard is used to guide development organizations through the functional safety aspects of electrical and/or electronic systems. Some of the key aspects that ISO 26262 tries to address are:

- The development process stages necessary for functional safety
- Guidance on the automotive safety life cycle
- Functional safety decomposition for systems engineering, hardware engineering, and software engineering
- Methods for using Automotive Safety Integrity Level (ASIL) standards to determine safety requirements for acceptable risk levels
- Guidance on acceptance criteria for validation activities based on ASIL

Development platforms for Model-Based Design such as MATLAB and Simulink enable you to create deployable algorithms for a variety of embedded systems. Simulink also enables you to verify those algorithms early and often in the development cycle. The IEC Certification Kit reference workflow uses these capabilities to provide a comprehensive workflow that you can use to create testable unit models, integration models, and system-level models. This high-level workflow is broken down into two sections as shown in Figures 1 and 2.



Note 1: You can omit SIL if you perform PIL with coverage analysis

Note 2: If you use a qualified compiler, PIL (or HIL) only needs to be done for Integration Testing and not for Unit Testing

Figure 1. Development phase 1: Model-Based Design.

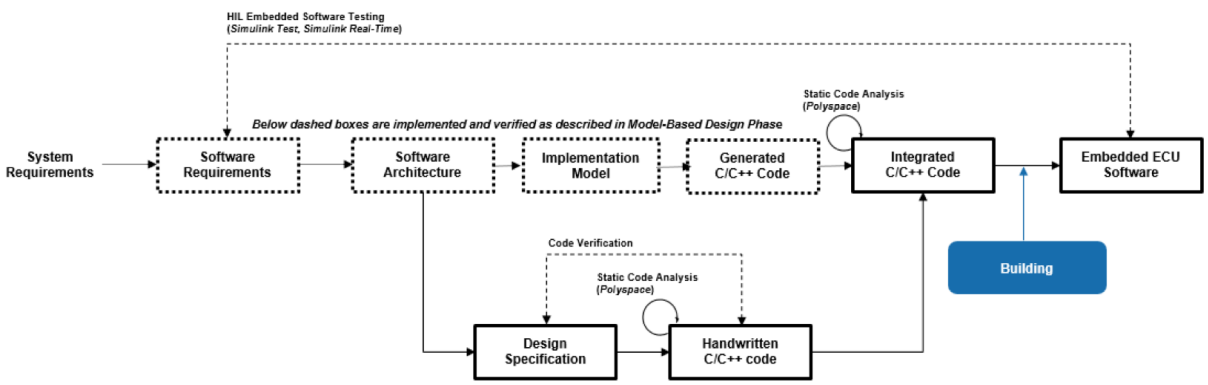


Figure 2. Development phase 2: Embedded software testing.

IEC Certification Kit stops short of providing suggestions on how a model architecture should be constructed. The general workflows shown above demonstrate at a high level the development and verification stages necessary for an algorithm. The amount of a work and the ease of following these steps can vary greatly depending on how an algorithm is broken down during the development and verification process. It is crucial to first think through these design choices for the algorithm's architecture because they can have a substantial impact on the efficiency of the development organization, reusability of the software, and testability of the software.

Best Practices for ISO 26262 Development with Model-Based Design

This white paper provides suggestions on modeling practices that you can use to segment algorithms to reduce verification and deployment efforts when adhering to ISO 26262 and using Model-Based Design. These best practices can be classified into the following categories:

- Model architecture:
  1. Use model reference for unit-level models
  2. Pick a strategy for grouping units into features
  3. Split ASIL and QM levels at the top level of the model
  4. Eliminate algorithmic content at the integration level
  5. Use model metrics to monitor unit complexity
- Signal routing and definition:
  6. Group bus signals by ASIL, feature, and rate
  7. Pass only necessary signals to units
  8. Optimize placement of signal and parameter objects
  9. Protect data exchanged between ASILs
- Code generation configuration:
  10. Determine a code placement strategy
  11. Use different name tokens for shared utilities

Please note that these best practices were designed to work together. Deviation from best practices while still achieving the overall objectives is possible, but not recommended.

Model Architecture

One of the first decisions when you are developing algorithms in Simulink involves the general model architecture. This is an important first step because the decisions made at this stage impact:

- Software testability
- Software reusability
- Unit and integration testing methods
- Ease of software integration
- Software segmentation for freedom from interference

These best practices, created by MathWorks consultants working with engineers who are targeting ISO 26262 compliance, will help you optimize these traits. The best practices center around areas that make ISO 26262 compliance easier while increasing efficiency in the validation, verification, and documentation phases.

1. Use Model Reference for Unit-Level Models

One of the main focal points in part 6 of ISO 26262 is the workflow for developing, validating, and verifying software units. Software units are intended to be the smallest testable parts of an application that must be individually tested for proper operation. Requirements will be mapped to and/or within these individual software units, and requirements-based test cases will be built to exhaustively test the software unit. Therefore, the modeling construct used for unit development needs to consider various aspects, including:

- Unit testability
- Unit code generation
- Unit complexity
- Unit testing workflow
- Unit traceability
- Unit documentation

These characteristics point to relying on model references as the primary modeling pattern for unit development. Model references have multiple characteristics that are conducive to the desired outcome, including:

- Code generation – There is a one-to-one mapping from model reference to source files generated.
- Reusability – Model references can be used in multiple places throughout an integration model.
- Testability – Model references are ideal for test harness construction.
- Team collaboration – Model references enable parallel development across developers and simplify the overall configuration management and version control processes.

Figure 3 shows how you can use model references to split functionality into testable units.

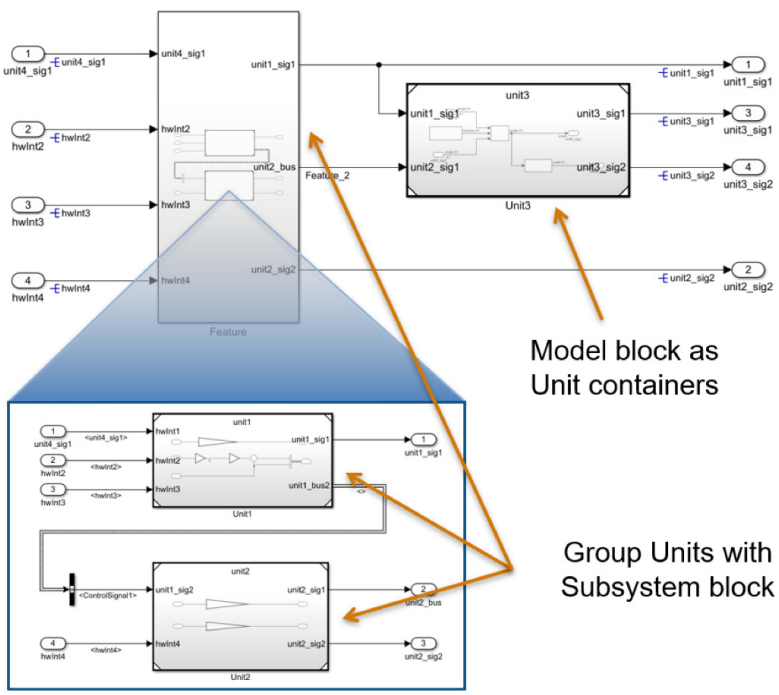


Figure 3. Using model reference for algorithmic units.

Although this approach is not recommended as a best practice, atomic subsystems in Simulink libraries can serve as software units. For this approach, developers need to explicitly manage and control the boundaries for partitioning and testing. For example, interfaces need to be locked down, meaning the user needs to specify the data types, sample times, and related block settings including code generation options such as function prototype control and file partitioning. Additional care and user assessment is required when placing an atomic subsystem in different execution contexts such as single rate, multirate, or multitasking. In Release 2019a, Simulink introduced library-based code generation for reusable subsystem libraries to facilitate such use of atomic subsystems. This feature offers new capabilities for better management, specification, generation, test, and reuse of atomic subsystems as standalone units.

You can also use subsystems as containers to further support functional grouping of similar units. Nested model reference is generally not recommended because it adds another layer of complexity in model and data management (see next best practice for more detail).

2. Pick a Strategy for Grouping Units into Features

When building large-scale models, you can choose from various types of modeling constructs to add model hierarchy, such as:

- Virtual subsystems
- Atomic subsystems
- Model references
- Library blocks

For ISO 26262, there is flexibility on how these unit models can be grouped under their respective ASIL. The previous best practice added one "hard" constraint, in that model reference should be used for the unit-level algorithm. However, grouping of units can be subsystems or even model reference. Some of the tradeoffs to consider are the number of model references that need to be managed and how firm a modeling boundary you want at a feature level. These units could be grouped into virtual subsystems if the architecture is indifferent on how the model is segmented and the user base is able to absorb working on a larger model. When you are determining a feature model segmentation strategy, consider:

- Generated code file and functionality grouping
- Parallel feature development by multiple developers
- The overhead associated with model reference files
- The ease of visibility for feature grouping in the design

At this intermediate level of the model between unit and the top level, the choice of modeling constructs is up to your organization's modeling guidelines and preferences. Figure 4 shows an acceptable method of using virtual subsystems.

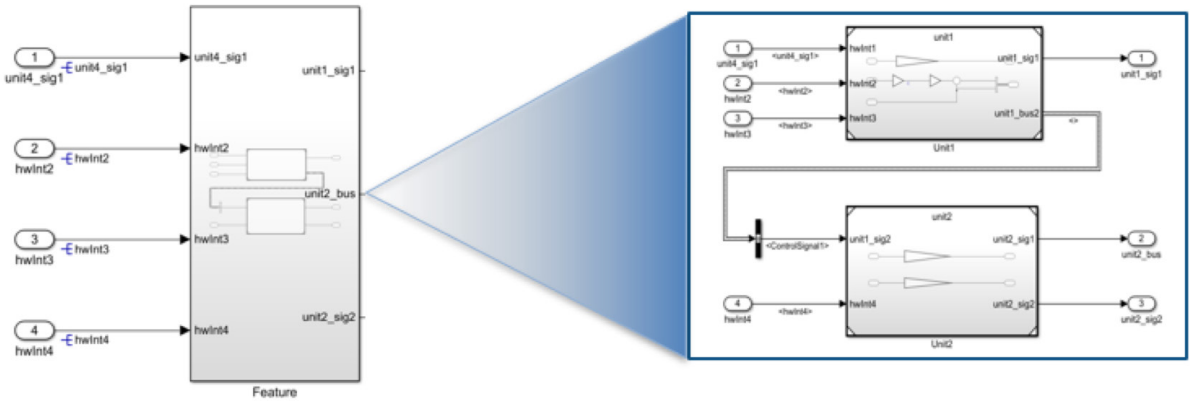


Figure 4. Grouping multiple units in a feature.

3. Split ASIL and QM Levels at the Top Level of the Model

One key concept in ISO 26262 centers around freedom from interference. ISO 26262 has five distinct safety levels (Quality Management [QM], and ASIL A–D) that can be used to classify system- and software-level functionality based on the functional safety aspects of the system. Electrical and electronic systems that are following ISO 26262 will likely have components at different ASILs. For example, a portion of an algorithm that reports out non-critical diagnostic data may be classified as a QM component, whereas a section of the algorithm that could impact the vehicle's ability to brake may be classified to a higher-level ASIL component due to the high degree of hazard/injury risk if a failure occurs.

A system with multiple ASIL components will benefit from an architecture that efficiently segments these algorithms into separate containers. The benefit will be seen for two reasons:

- Each ASIL can have different development, validation, and verification requirements.
- Separating and segmenting ASILs enables freedom from interference.

Since the various ASILs will be split, you should choose a modeling construct to aid in that segmentation. Use model references so that when the algorithm is deployed, there will be a firm boundary at each ASIL's border. Therefore, you should split the top level of the system-level model into multiple model references with each model reference representing a separate ASIL. Note that in this case, due to code generation configuration settings (see the Code Generation Configuration section for detail), the system-level model is for simulation only, and code generation can only be done separately based on ASIL. Generating code at the unit level and then integrating it is another option. However, generating code at as high a level as possible reduces the amount of overhead during code integration.

By using a model reference for each ASIL or even more generically, whenever freedom from interference is needed, separate functions and source files will be generated for each model reference. By following this and the code generation configuration best practice, each segmented partition will have its own source files, shared utilities, and data definitions, which will make it easier to achieve freedom from interference between the different sections of the algorithm. Figure 5 demonstrates how a model could hierarchically be split based on ASIL.

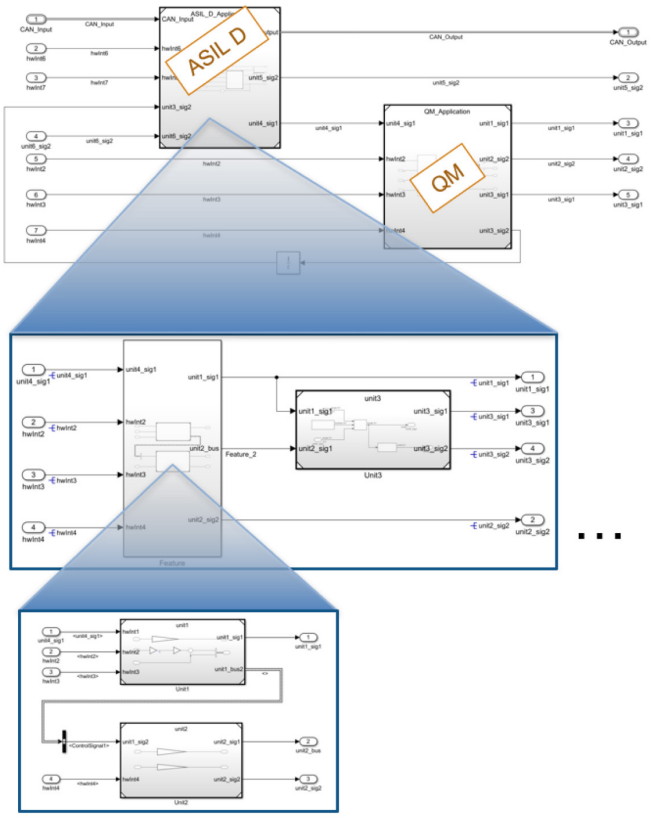


Figure 5. Model hierarchy based on ASIL.

4. Eliminate Algorithmic Content at the Integration Level

ISO 26262 has the notion of multiple testing levels in the representative architecture, including unit level, integration level, and system level. Typically, a software unit must go through various levels of testing rigor based on the targeted ASIL. For example, ASIL D may require full modified condition and decision coverage (MCDC), whereas for ASIL A or B, condition and decision coverage may be acceptable. Because of this, units are the only place where algorithmic functionality should be implemented. If algorithmic content occurs at the integration or system level of the model, it can be more difficult to achieve full coverage of the design. For this reason, it is recommended to ensure that no algorithm content occurs outside of unit-level models (Figure 6).

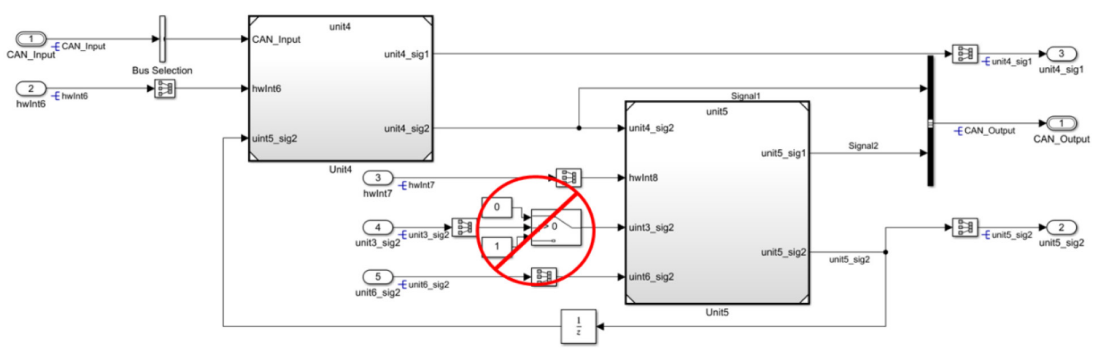


Figure 6. Avoiding algorithm content outside of the unit.

This best practice can also be mapped to MathWorks Automotive Advisory Board (MAAB) rule db_0143: Similar block types on the model levels.

5. Use Model Metrics to Monitor Unit Complexity

Many organizations realize late in the development cycle that their algorithms will be difficult to validate to the level of coverage that ISO 26262 recommends. This typically stems from the lack of architectural consideration at design time and management of unit-level size and complexity during development. One methodology that can alleviate this issue is to set unit size metric thresholds for the entire development organization. These thresholds can include:

- Maximum number of inputs and outputs per unit
- Reusable libraries
- Cyclomatic complexity
- Number of elements

To manage the unit's complexity, include the review of these metrics during the model review process of the development cycle. This will enable visibility of the complexity and scope of how the algorithm has been designed during the development process. The Model Metrics Dashboard in Simulink Check™ (Figure 7) can make this process easier.

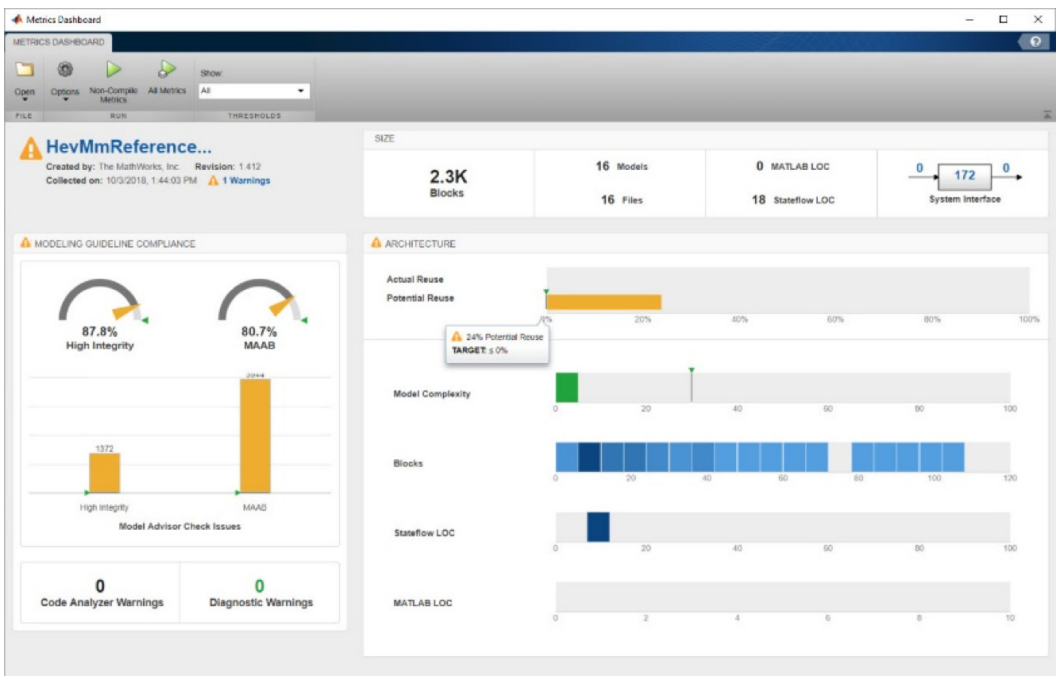


Figure 7. The Model Metrics Dashboard in Simulink Check.

The Model Metrics Dashboard can make total block count, MATLAB lines of code (LOC), Stateflow® LOC, model complexity, number of blocks, and other metrics easily visible to the modelers and model reviewers. The dashboard supports the design workflow by continuously monitoring models so you can ensure that they are not being divided into units that are too small (increasing management complexity) or too large (unable to be easily tested and difficult to reuse).

Threshold values are often an area of debate. This is because the model complexity usually can't be measured from a single aspect of the model, and often requires you to analyze multiple metrics to make meaningful decisions. For example, one Stateflow chart can have very high cyclomatic complexity while its block count is only one.

The recently published paper ***Model Quality Objectives for Embedded Software Development with MATLAB and Simulink***, from MathWorks and a group of automotive companies (Delphi Technologies, Bosch, PSA, Renault, and Valeo), provides guidance on metrics and thresholds. For example, model cyclomatic complexity should be 30 or lower, and the number of model elements should be less than 500.

Each company will have different threshold values, and MathWorks Consulting Services provides support in this area.

Signal Routing and Interface Definition

Part 6 of ISO 26262 has multiple considerations that address interface complexity and data exchange between units and components. For example:

- Table 3 - 1b: Restricted size and complexity of software components
- Table 3 - 1c: Restricted size of interfaces
- Table 7 - 1g: Data flow analysis
- Table 7 - 1k: Interface tests

To achieve these goals, it is important to determine architectural strategies for how data will be exchanged between units, components, and different ASILs. While working with various engineering teams, MathWorks has created multiple best practices to reduce the work necessary to analyze data interface exchanges between unit level models, components, and various ASILs. This section provides four best practices aimed at managing interface complexity and data exchange.

6. Group Bus Signals by ASIL, Feature, and Rate

ISO 26262 Part 6 Table 7 - 1g recommends that development teams perform data flow analysis. Data flow analysis is necessary to understand how signals are passed through a software algorithm and down to the unit level. This type of analysis can spot areas where signal requirements clash or where various signals should not be directly used based on characteristics of the provider. To perform this analysis, organizations must develop a bus hierarchy strategy to make it easier for the developer to understand where the signal is coming from and what the characteristics of the signal are. If you don't specify how bus signals should be grouped hierarchically, the following issues can occur:

- Inefficient bus segmentation and modeling patterns
- Inconsistent bus grouping from developer to developer
- Modeling difficulty when splitting and recreating buses
- Inefficient code generation

Just as model architecture requires a top-down design approach, the same is true for bus hierarchy. To manage the ASIL for each signal, group signals based on task rate and ASIL in the bus hierarchy (Figure 8). By grouping these signals based on ASIL, it will be easier to determine the provider of a signal and determine if the signal is being used in a unit of a higher ASIL. For example, if a signal is provided from a QM component but used by an ASIL component, additional analysis will be necessary to ensure that the dependency on this signal has been analyzed and accounted for.

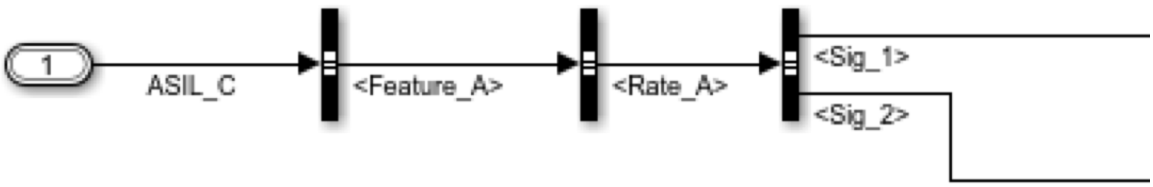


Figure 8. Grouping signals in a bus hierarchy.

7. Pass Only Necessary Signals to Units

Table 3 and Table 7 of ISO 26262 Part 6 have important suggestions relating to unit-level interfaces and data exchange. These two tables mention that the size and complexity of software components and interfaces should be reduced where possible. Also, during the verification process, users should perform interface tests. If unnecessary variables are passed down to a unit-level model, additional testing will be necessary to ensure that these signals do not have an impact on the unit. To alleviate this concern, reducing the inputs that are passed to the unit level can help. To accomplish this, you can split bus signals prior to entering a unit-level model to only signals used in the corresponding unit (Figure 9).

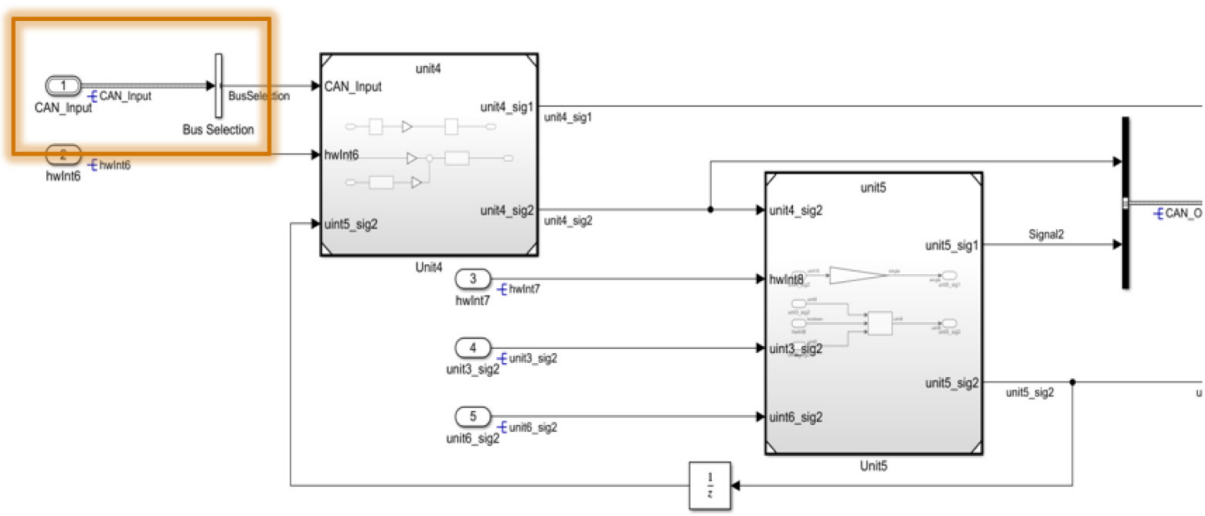


Figure 9. Splitting bus signals entering a unit-level model.

8. Optimize Placement of Signal and Parameter Objects

The high-level use case for signal and parameter objects is to define the interface between the model and base software. In such a use case, parameter objects are typically used for specifying calibration values and placed inside model blocks such as Gain and Lookup Table blocks. The usage of signal objects is more complex, but it usually is associated either with internal signals to support calibration activities or with root input and output ports (Figure 10).

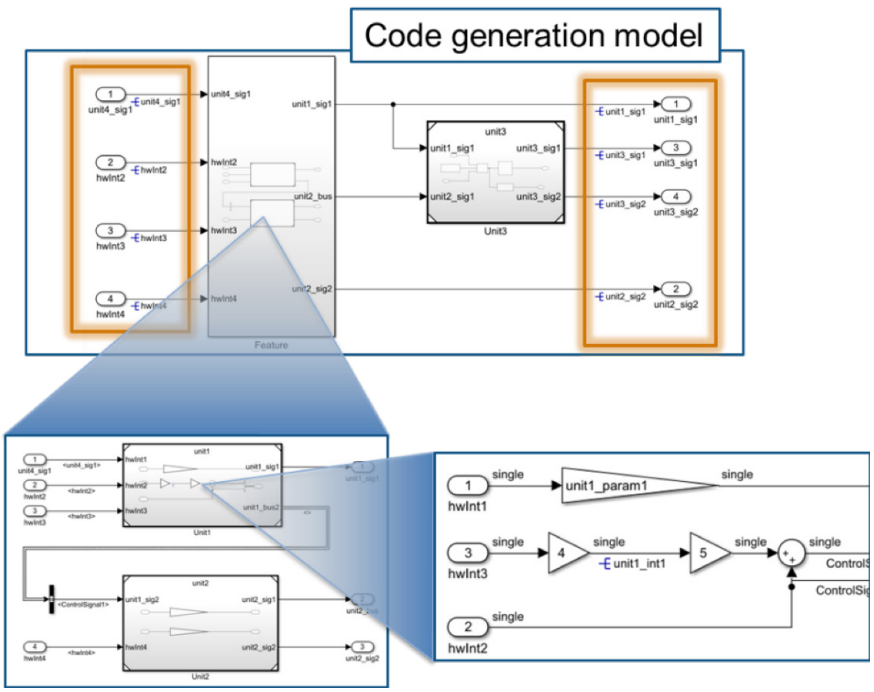


Figure 10. Placing signal and parameter objects.

It is important to note that the interface to the model reference block does not contain signal objects. This is because signals at the boundary of the unit level are considered internal interface signals. This also assumes that the code is being generated at the highest ASIL partition as mentioned in best practice #6 above. For the internal interface signals, it is best to let Embedded Coder® define those signals based on its internal optimization algorithms. Data type information, however, does need to be explicitly defined as part of the port block configuration at model reference boundaries, as shown in Figure 11.

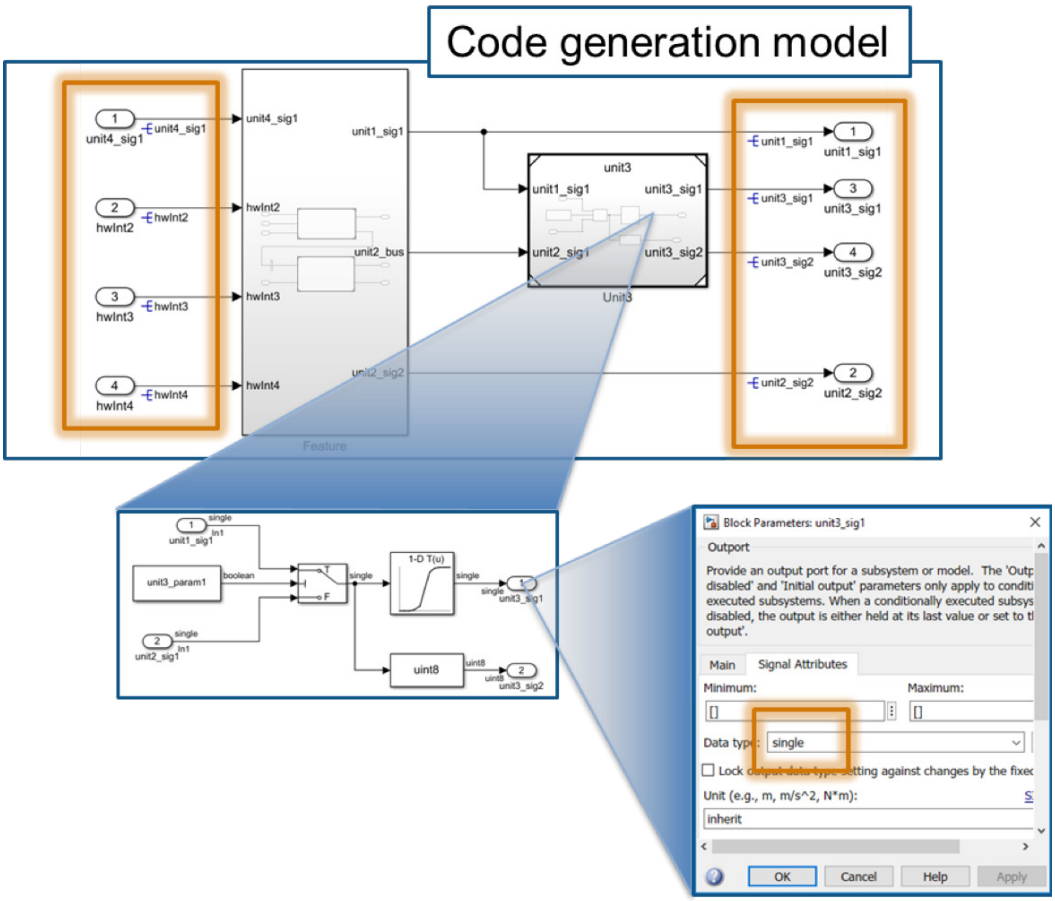


Figure 11. Defining data type for a port.

This best practice restricts the placement of interface signal objects to the code generation model boundaries. It also ensures that when code is generated for each ASIL, the root-level output ports will point to the corresponding interface function in the other ASIL section.

The storage class used for the signal and parameter objects can be set based on the software architecture and coding practices. The exception will be the protection needed for data exchange between ASIL models or partitioning as required by freedom from interference.

9. Protect Data Exchanged Between ASILs

One consideration when code for each ASIL is generated separately is that a protection method is needed to exchange data between each ASIL. Multiple strategies exist using storage classes on the ASIL sections' root-level input and output ports. One prevalent method is to use a storage class that has get and set access functions for the data. By using get and set access functions, you can add additional protection to these interfaces so that only appropriate software components are accessing data.

Figure 12 depicts one example of how GetSet storage classes can be used on the root-level input ports and the corresponding generated code.

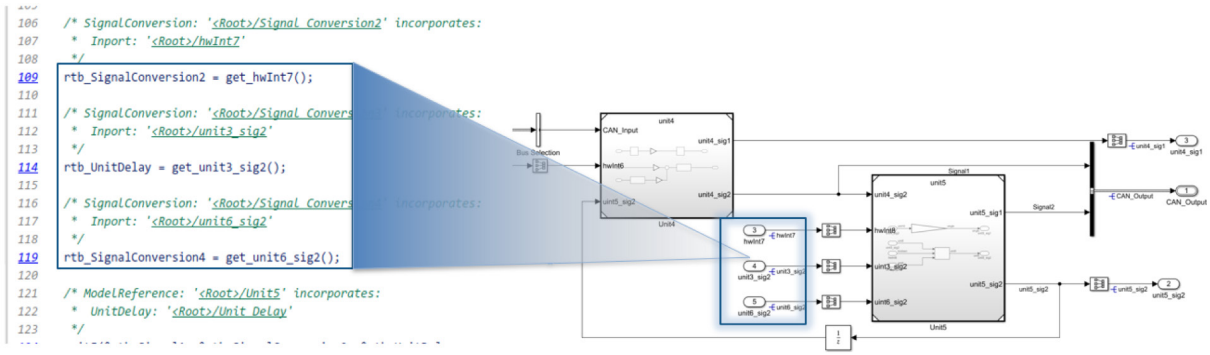


Figure 12. Non AUTOSAR Interfaces Between ASIL Components

Figure 13 shows how one of the storage classes was configured to ensure that the Get and Set API match between ASILs.

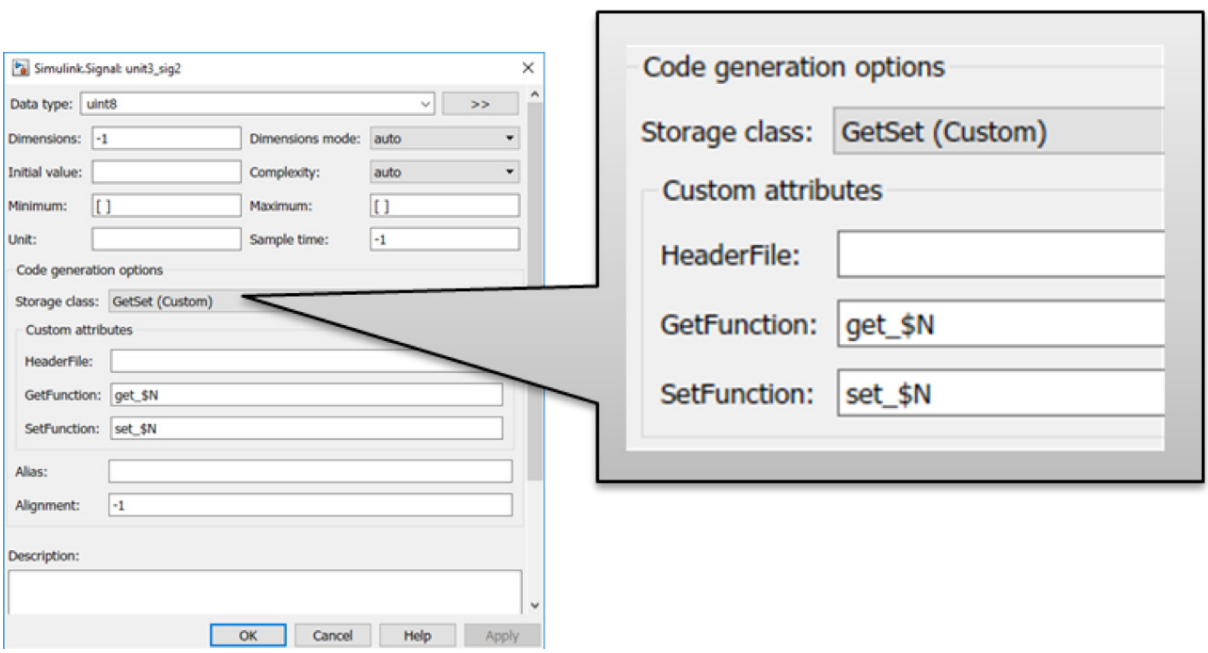


Figure 13. Configuring code generation options for the root-level I/O storage class.

It is important to note that the GetSet storage class objects only provide an entry point for the interface protection. The actual implementation of the protection is typically done through low-level software layers implemented by hand coding, which can be easily integrated using the GetSet storage class.

Code Generation Configuration

Once the model has been developed according to the best practices listed above, there are still code generation configuration settings that need special attention to achieve objectives set forth by ISO 26262. This section provides best practices for code generation configuration setting with respect to code placement and separation of shared utility files. Again, the configuration listed below assumes the previous best practices have been followed.

10. Determine a Code Placement Strategy

One key design concept in ISO 26262 is freedom from interference. Freedom from interference is necessary to ensure that if there is an issue with one section of the system at a particular ASIL, it will not impact functionality at a separate ASIL. For example, if a QM or ASIL A component has an issue or a failure occurs, the design should segment this functionality away from functionality that is ASIL D to ensure that the ASIL D functionality can continue operating. In embedded systems, one type of fault that is a concern is if portions of the application have access to sections of memory or functions that they should not have access to. One way to address this concern is to separate the functionality into separate memory sections or on separate cores of a microprocessor. This can be done by telling the compiler into which section each variable, function, or file should be placed. Figure 14 demonstrates how the application could be separated.

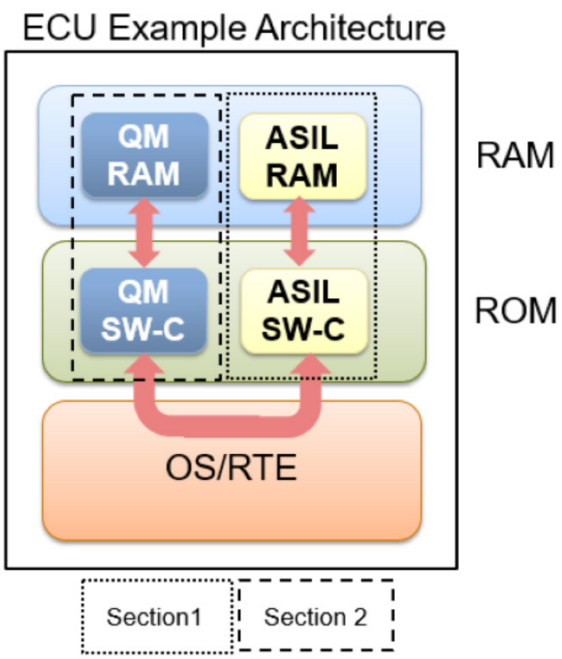


Figure 14. Segmenting the system architecture to ensure freedom from interference.

Another technique is to split various ASIL and QM levels into separate memory sections. Splitting the memory sections alleviates some concerns with various ASILs unintendedly interacting with each other, for example, a function within the QM software component (SW-C) writing to a protected ASIL RAM location. To configure this, memory sections can be selected in the configuration options as shown in Figure 15. This method does not have to be used, but it does simplify the overall workflow.

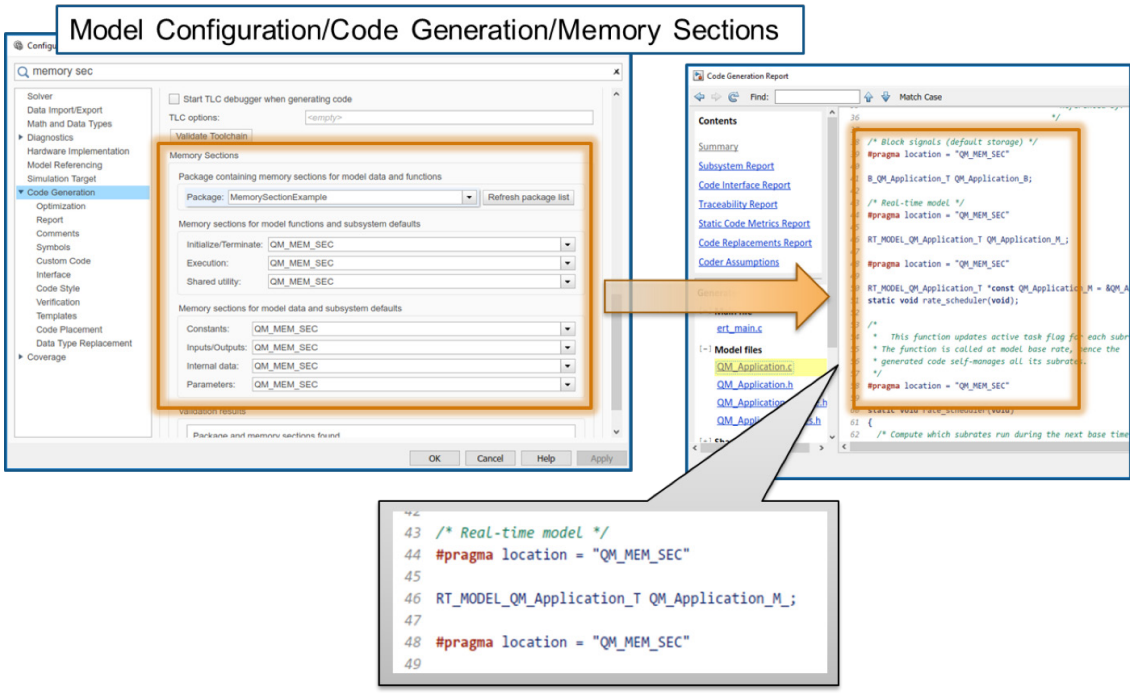


Figure 15. Configuring parameter settings for memory sections.

11. Use Different Name Tokens for Shared Utilities

Embedded Coder generates common utility files knows as shared utilities. These files are basic functions that are used across the code generation model. This method presents an issue in a safety-related application because there will not be any distinctions between the sources of the shared utility files. To compile code into a signal application with the segmentation concept, shared utilities must be generated and allocated based on their ASIL. This can be done through code generation configuration, as shown in Figure 16.

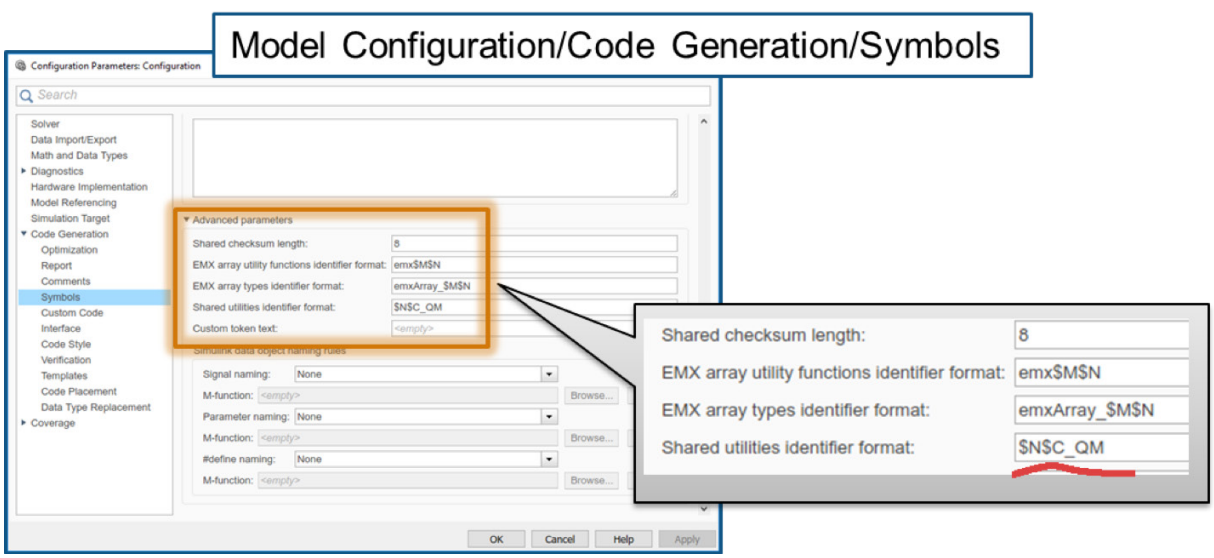


Figure 16. Configuring shared utility settings.

With the above settings, you can create the generated shared utility with a unique identifier that is a function of the ASIL. For example, the shared utility for the Lookup Table block when configured based on the above QM suffix is shown in Figure 17.

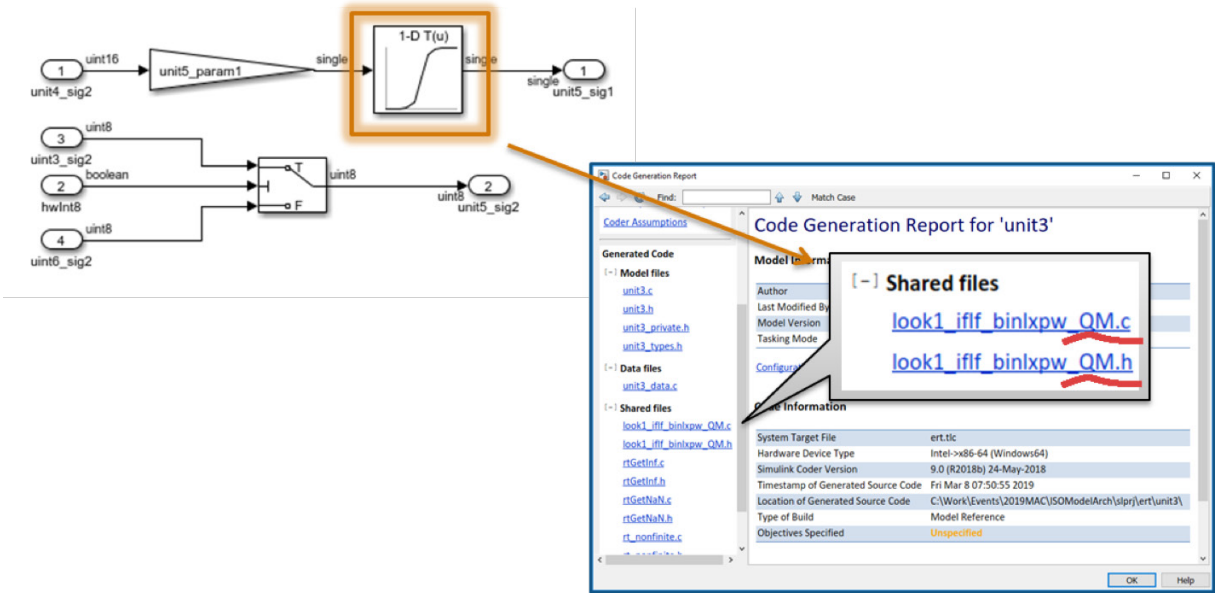


Figure 17. Shared utility configured for QM.

Summary and Future Work

The findings presented in this document are best practices created through multiple MathWorks consulting engagements. These best practices are proven enablers to adoption of ISO 26262. However, following these best practices does not guarantee ISO 26262 compliance because they address a subset of all ISO 26262 requirements, and each application has its unique needs.

Future work is underway in applying the above best practices for AUTOSAR standards. ***Request an early draft of the paper.***

Learn More

- ***ISO 26262 Support in MATLAB and Simulink*** - Overview
- ***ISO 26262 Process Deployment Advisory Service*** - Consulting Services