

## MODEL-BASED DESIGN FOR LARGE HIGH INTEGRITY SYSTEMS: A DISCUSSION ON DATA MODELING AND MANAGEMENT

Mike Anthony<sup>\*</sup> and Matt Behr<sup>†</sup>

One of the most important concepts in Model-Based Design is that of the model as an executable specification. Building large models for the generation of production-quality embedded software requires the development of a modeling style that guides and enforces model architecture, interface definition, modeling standards, and data management. This paper focuses specifically on data management with Model-Based Design using MATLAB and Simulink. Models necessarily rely on external data and functionality to create an environment that allows initialization, trim, linearization, simulation, analysis, and code generation. This paper describes the fundamentals of how to define and manage parameters and signals within a model. It also discusses the implications of data management style on componentization, flexibility, readability, and code generation. Where relevant, recommendations suited for models targeting embedded code generation for mission-critical and high integrity systems are highlighted.

### INTRODUCTION

One important concept in Model-Based Design is that of the model as an executable specification. “Executable” implies that the model exists as more than just a document, but as a functional part of the design process. “Specification” implies that the model, aside from being an environment in which an algorithm is developed and tested, serves as documentation of the algorithms. This builds on the discussion in *Model-Based Design for Large Safety-Critical Systems: A Discussion on Model Architecture*<sup>(1)</sup>. However, as much as the idea of the model as a single source of truth influences decisions related to model architecture, the model itself is only one part of the total modeling environment. The model will necessarily rely on external data and functionality to create an environment that allows initialization, trim, linearization, simulation, analysis, and code generation. There are many ways to express data in MATLAB and Simulink, just as there would be in any software environment. The question, then, is what are the implications of these different approaches?

The choices made regarding data management will affect the componentization, flexibility, and readability of both the model and the generated code. The impact of data management on all of these topics will be discussed. In this paper, particular emphasis will be placed on recommendations that specifically apply to large models targeting embedded code generation for high-integrity systems. For these applications the ideal data management scheme is that which most closely expresses the desired implementation of the data in the embedded system. Clearly this leaves room for much debate, as there are many opinions on how best to implement data in

---

\* Application Engineering, The MathWorks, Inc., 3 Apple Hill Drive, Natick, Massachusetts 01760-2098, U.S.A.

† Aerospace Industry Marketing, The MathWorks, Inc., 3 Apple Hill Drive, Natick, Massachusetts 01760-2098, U.S.A.

an embedded system. Which of these is best is inconsequential to this discussion, as the only important consideration is that the model reflects whatever decision is made for the embedded system in order to maximize traceability and determinism.

A discussion of the fundamentals of data is the logical place to begin. Data will be defined as all numerical information related to the model. This includes signals, parameters, lookup table data, test vectors, simulation results, flight-test data, etc. The following discussion will be constrained to the definition of parameters and signals used in a Simulink model. Other resources are available that discuss managing all of the data and files related to a Simulink model.<sup>(2)</sup>

The paper will first define signals and parameters and outline the fundamental differences between the two. Both signals and parameters will then be discussed in more detail. Simulink provides flexibility in terms of how both signals and parameters are stored, defined, and managed. The impact of different approaches on componentization, readability, and code generation will be highlighted. Finally, some example modeling styles will be discussed which combine the use of signals and parameters to achieve different objectives.

## PARAMETERS AND SIGNALS

For clarity, the following discussion assumes the term signal to mean data represented by a line in the Simulink model, and a parameter as data represented by a numerical value or workspace variable provided to the dialog window or mask of a Simulink block. With these definitions in mind, a fundamental difference between signals and parameters is apparent. Signals can vary with time as a result of input changes or the operations of blocks preceding them in execution. Parameters are defined during initialization and, unless explicitly changed by the user, remain constant. It would seem, then, that the architectural decision regarding signals and parameters is readily apparent. If the data varies with time, it should be a signal. Otherwise, it should be a parameter. Indeed, this is a common approach and should be kept in the back of one's mind while developing a modeling style. However, in many cases there is flexibility on whether data is defined as a parameter or signal in a model.

### Simulink Parameters

A parameter is a way to provide a value to a portion of the algorithm represented by a Simulink block or subsystem. For example, when creating a model, it is possible to set the "Constant value" parameter of each instance of the Constant block separately so they each behave differently. This holds for built-in blocks such as the Constant block, subsystems, and model references. An aerospace example of parameterized data is aerodynamic look-up tables or controller gain scheduling data.

Simulink evaluates parameters before running a simulation. Each value remains constant during execution unless a parameter is declared as tunable. A *tunable parameter* is one whose value can be changed without recompiling the model. Parameters are tunable by default in Simulink. Tunability of the generated code can be controlled via the *Inline parameters* option in the Simulation Parameters dialog box. A user must give thought to the desired tunability behavior during simulation and code generation. Simulation parameters, data definition, and code generation options will have effects on parameter definition and tunability in the generated code. In general, tunability in the generated code provides flexibility to change parameters in the generated code without requiring regeneration of code from the model or recompilation of the software project.

The ways to define Simulink parameters are hard-coding them into parameter dialogs or defining them as workspace variables. Several options exist for defining them as workspace variables. Simulink data objects can be created to capture additional parameter information.

Users can also use the base workspace or model-specific workspaces as location to define parameters, depending on the desired scope or visibility of that parameter.

How a parameter is defined will affect the readability, modularity, and flexibility of the model as well as the generated code. These considerations are taken into account in the following overview of the most common methods for defining parameter data.

### Hard-Coding Parameter Values

The meaning of numerical values in this case is the use of the actual number in the mask of any Simulink block. During initialization, Simulink parses the text entered into the mask and resolves it to a numeric value. An example where the gain values are parameterized using numerical values is shown in Figure 1 below.

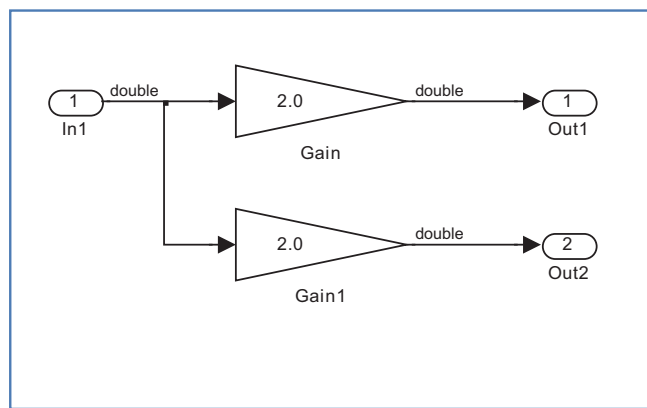


Figure 1: Model using numerical parameters

Numerical values are the most readable and modular way to represent data in Simulink. Using numerical parameters allows the model to be a self-contained, fully-specified description of the algorithm. By defining parameters in the model, dependencies on additional data files are completely eliminated. This is advantageous from the perspective of configuration management, as there is a single source of truth for both the algorithm and the data. However, for the very reasons that numerical parameters provide readability and modularity, this method is also the least flexible approach in defining Simulink parameters. In order to change a parameter value, the user must navigate to the block and change its value. While this can be automated via scripts or functions, these utility functions then also require maintenance.

The tradeoffs and effects on the model between different methods for defining parameters is quite straightforward. The same is not necessarily true concerning code generation. As such, the code generation impact will be discussed in more detail, as this will not only provide an understanding of each parameter definition method, but also shed some light upon possible best practices related to controlling the style of the generated code. One may think that using numerical values as parameters would automatically result in numerical values being “inlined” in the generated code. However, even in this simple case, consideration must be given to code generation options. Code is generated twice from the model in Figure 1, changing the *Inline parameters* option, as shown below:

With *Inline parameters* enabled:

```
void Inline_Example_step(void)
{
```

```

Inline_Example_Y.Out1 = 2.0 * Inline_Example_U.In1;
Inline_Example_Y.Out2 = 2.0 * Inline_Example_U.In1;
}

```

With *Inline parameters* disabled:

```

void Inline_Example_step(void)
{
Inline_Example_Y.Out1 = Inline_Example_P.Gain_Gain * Inline_Example_U.In1;
Inline_Example_Y.Out2 = Inline_Example_P.Gain1_Gain * Inline_Example_U.In1;
}

```

The code generated with the *Inline parameters* option enabled most closely mimics what is expressed in the model. This is extremely desirable. Recall the discussion on Model Architecture <sup>(1)</sup>, in which a key practice is to model in an intuitive manner. In other words, if there is a common expectation that the model or code generation process will behave in a certain way, the best practice is to ensure that the model conforms to that expectation. For the purposes of this paper, this shall be called in the “intuitive rule.” In this case, by using numerical parameters in the model, a common expectation would be for those parameters to appear as numerical values in the generated code. As shown above, this can be accomplished by enabling the *Inline parameters* option. Thus, a possible modeling rule related to the intuitive rule might be: if numerical parameters are used, ensure that the *Inline parameters* option is enabled.

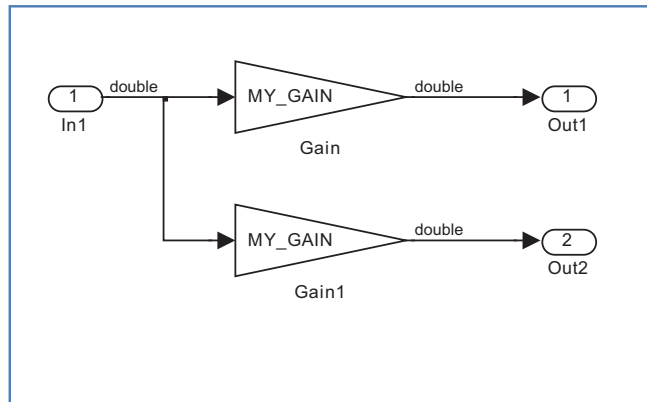
### Defining Parameter Values via Workspace Variables

When working in Simulink, the easiest way to think of the MATLAB base workspace is as a repository for global data. Any variable defined in the base workspace is available for use at any location within the Simulink model, regardless of hierarchy, boundaries, etc. In many ways this is exactly the same behavior as global data in software.

While the base workspace is the most common repository for variables, each model also has a model workspace. The model workspace is scoped such that any data defined in the model workspace is available only to models and blocks inside that model. The C equivalent to this would be local data, declared within the function call. An example using the model workspace is provided later.

When Simulink encounters a variable name during compilation, it first tries to resolve the variable in the mask workspace, if a mask exists. If not resolved there, Simulink then looks in the model workspace. If the model workspace is empty or the variable is not defined there, Simulink then looks in the base workspace. While the model workspace is a useful tool in that it allows data to be defined per model, it should be noted that some flexibility is lost as the data must be altered either in the Model Explorer or via special MATLAB commands. Variables cannot be loaded or accessed as easily in the model workspace as when they are defined in the MATLAB workspace. As with numerical parameters, this lack of flexibility is purposely related to the additional determinism offered by this method. Also note that only one model workspace exists even if multiple instances of the same model are instantiated via model reference blocks. The ability to have multiple instances of such a model making use of different data is typically accomplished via masks. This idea is covered in more detail in the examples of combining signals and parameters later.

Understanding the basics of the model and base workspaces, the next method of parameter definition is the use of workspace variables. This approach takes advantage of the inherent relationship between Simulink and MATLAB. An example of a model with workspace variable parameter definitions is shown in Figure 2. It uses the same model as Figure 1, but the gain value is defined as MY\_GAIN, a workspace variable.



**Figure 2: Model using workspace variables**

Using workspace variables is by far the most flexible option for defining data. Multiple blocks within a Simulink diagram can refer to the same MATLAB variable. If changes need to be made, this variable can be changed and Simulink will use the new value throughout the model. This effectively decouples the data from the model. However, by doing so, the model is no longer fully self-contained. This method creates a dependency on the MATLAB workspace. Also, variables will likely be stored in a .m file or MAT files, which must be maintained and version controlled. A rigorous version control system must be in place to ensure that the proper version of the data is loaded with the corresponding model for proper simulation results.

This version control system is also necessary for model reviews, as readability of the model is negatively affected by the use of workspace variables. Consider the differences between Figure 1 and Figure 2. Examination of the model shown in Figure 2 does not provide a complete understanding of the algorithm. Specifically, if a team were to review this model, the functionality would be unclear because of the variable definition of `MY_GAIN`, whose value is not apparent. When using numerical parameters, one can export the model to a Simulink Web view and obtain a fully specified, hierarchical design document that is completely self-contained and independent of any other files or MATLAB licenses. This is highly useful for model reviews. The use of workspace variables can make this task more challenging. That said, there is a method using block annotations with the style shown in Figure 2 to achieve the same clarity. This method can be accomplished via the Block Annotation pane of the block properties dialog for any block. There is also a capability in Simulink to automatically generate a System Design Description. In the case of workspace parameter definitions, this report will gather the parameter values currently defined in the base workspace and include those values in the report. This report is more complete than the Simulink Web view, but is also less interactive and usually much longer for models of appreciable size.

After understanding the effect of workspace variables on the model, the next step is to consider the effect of code generation. Recall the “intuitive rule” as guidance for selecting the appropriate code generation options, as using the default code generation options when defining parameters as workspace variables can initially lead to non-intuitive results. The following example will highlight how Real-Time Workshop Embedded Coder deals with parameters defined as workspace variables.

Code generated from the model in Figure 2 with the *Inline parameters* option enabled is below:

```
void Inline_Example_step(void)
{
  Inline_Example_Y.Out1 = 2.0 * Inline_Example_U.In1;
  Inline_Example_Y.Out2 = 2.0 * Inline_Example_U.In1;
}
```

```
}
```

Perhaps unexpectedly, this is equivalent to the code generated from the model in Figure 1 with the *Inline parameters* option enabled.

The generated code from the model in Figure 2 with the *Inline Parameters* option disabled is as follows:

```
void Inline_Example_step(void)
{
  Inline_Example_Y.Out1 = Inline_Example_P.Gain_Gain * Inline_Example_U.In1;
  Inline_Example_Y.Out2 = Inline_Example_P.Gain1_Gain * Inline_Example_U.In1;
}
```

Again, this is the same as the code generated from the model in Figure 1 with the *Inline parameters* option disabled. The code generated with the *Inline parameters* option disabled also shows a perhaps unexpected inconsistency between the model and the generated code. In the modeling environment, a single variable was defined and used in two locations. Recalling the “intuitive rule,” what is the expected appearance of the code for the model in Figure 2 with the *Inline parameters* option disabled? Perhaps the following:

```
void Inline_Example_step(void)
{
  Inline_Example_Y.Out1 = MY_GAIN * Inline_Example_U.In1;
  Inline_Example_Y.Out2 = MY_GAIN * Inline_Example_U.In1;
}
```

Indeed this would be extremely intuitive, especially given the behavior seen with the model in Figure 1. The code generated with the *Inline parameters* option disabled is perhaps a bit closer to the expected behavior, in that it uses variable names. However, there is a fundamental difference. In the generated code, the gain value is stored in two separate variables, even though it is the same value and is defined by the same workspace variable in the model. This is certainly a divergence from the expected behavior. Furthermore, the actual generated code is less memory-efficient than the expectation, as the same parameter value will be declared for each instantiation, rather than reusing a single variable each time. The more often the parameter is used, the higher the memory cost.

Thinking of the generated code for a larger system, it is almost certain that some parameters will be reused in many places throughout the algorithm. Commonly reused parameters include gravity, radius of the earth, mass of the earth, or many unit conversions. For such cases, it is highly desirable to define these parameters in a single place and have all instances of that data in the algorithm refer to that single definition. This shall be called single-source parameter definition. This prevents conflicting definitions of the same parameter in different parts of the code.

In the code environment this equates to a global variable definition, as in the expected code generation output above. The default code generation options, however, do not mimic the single-source parameter definition behavior in the model. There is a method to generate code that conforms to the expected output. This is accomplished through data objects.

### Defining Parameter Values via Simulink Data Objects

Data objects are simply workspace variables, with a much more rigorous definition than a basic workspace variable. Basic workspace variables, when used in Simulink, are implemented as Simulink data objects, such as `Simulink.Signal` and `Simulink.Parameter` objects.

These definitions of data allow only a user specification of the value of the data. Defining variables as custom data objects allows a user to explicitly set not only the value of a workspace variable, but its description, units, data type, and storage class, among other properties. In addition, custom properties can be defined. Recall the model in Figure 2. When using data objects, the model will appear exactly the same. As such, in terms of modularity and readability, Simulink data objects are equivalent to normal workspace variable definitions. In some ways they are more flexible in that the additional information they contain can be used to specify data structure. For example, all the constant values used throughout the model could be grouped within a constant structure in the generated C code by properly specifying their storage class.

The impact of using data objects is much more clearly observed during code generation. Consider again the model in Figure 2. Using this model, but defining an `mpt.Parameter` object (rather than the default `Simulink.Parameter` object) for `MY_GAIN` and enabling the *Inline parameters* option, the following code is generated:

```
void Inline_Example_step(void)
{
  Inline_Example_Y.Out1 = MY_GAIN * Inline_Example_U.In1;
  Inline_Example_Y.Out2 = MY_GAIN * Inline_Example_U.In1;
}
```

This is exactly the same as the expected output from earlier. Furthermore, this expected output is an intuitive result from the model in Figure 2. In both the model and the generated code, single-source parameter definition is implemented.

Note that this approach aligns extremely well with a data dictionary coding style. Thus, another possible modeling rule if the desired coding style is a data dictionary approach, is that parameters in Simulink should be defined as variables in the base workspace. Furthermore, these variables must be defined using data objects such that the generated code is an intuitive output of the model.

### Parameters Summary

As described above, all of the methods for defining parameters have advantages and disadvantages. Using workspace variables requires object definitions and careful use of the workspace to ensure consistent simulation results while also making the model slightly less self-contained and fully specified. Workspace variables and a data dictionary style approach are easier for applications where tunability of parameters in the generated code is desired. However, for high integrity applications, additional consideration should be given to tunability. In such an application, changing a parameter value in the generated code should only be done if the proper analysis and regression testing has been successfully completed to justify that change and ensure that the design is still safe. When using Model-Based Design, this typically means adjusting the parameter values in the model and fully understanding the impact of that change in the modeling environment.

### SIMULINK SIGNALS

The above section explained several options for defining parameters. The other type of data in Simulink is a signal. A Simulink signal is a line connecting two or more Simulink blocks. This section will discuss the concepts of signals and parameters, and differentiate between the two. Obviously treating data as a signal or a parameter will significantly affect how it is implemented in the model. Thus, it must be clearly defined as part of the model architecture and style guidelines.

### Choosing Parameters or Signals

A fundamental difference between signals and parameters is that signals can vary with time whereas parameters will be constant within a given simulation run. It would seem then that the intuitive architectural decision regarding signals and parameters is clear. If the data varies with time, it should be a signal. Otherwise, it should be a parameter. Indeed, this is a common approach. However, this is not necessarily always the optimal solution. The code implementation of the algorithm must also be considered.

The concept of numerical versus variable parameter definitions and the impact on both the look of the model as well as the generated code was discussed earlier. The following example examines a similar issue, with the additional complication of using a signal or a parameter to express the gain. Figure 3 uses a numerical parameter in a Constant block, and then routes that value via signals in the model.

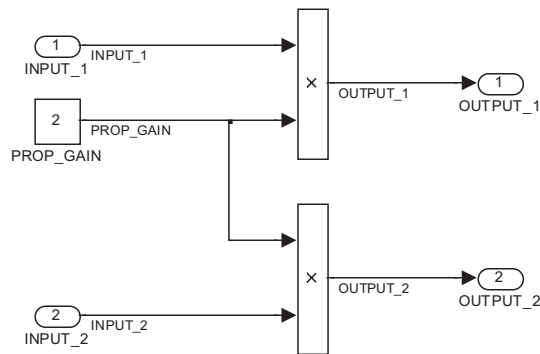


Figure 3: Constant block with numerical parameters

Figure 4 is functionally equivalent to the model in Figure 3, but uses a workspace variable parameter implemented via a Gain block.

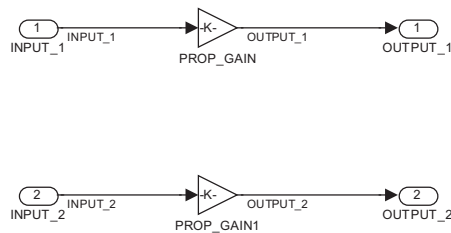


Figure 4: Workspace variable parameters

Assuming the best practices noted in the previous section were followed, and the variable `PROP_GAIN` was defined by an `mpt.Parameter` object, the function generated from the model in Figure 4 (`ParameterGain.mdl`) is just as expected from the previous example.

```

void ParameterGain_step(void)
{
ParameterGain_Y.OUTPUT_1 = PROP_GAIN * ParameterGain_U.INPUT_1;
ParameterGain_Y.OUTPUT_2 = PROP_GAIN * ParameterGain_U.INPUT_2;
}

```

In Figure 3, the proportional gain is defined via a numerical parameter in the Constant block, which is then applied to the two input signals via product blocks. In this style, the Constant block serves to essentially convert the parameter into a signal. As with the first example, the advantage to this approach resides mostly in the clarity and modularity. By using a single Constant block to define the parameter and then supplying that parameter to other blocks as a signal, the dependency on that constant value is clear. In this example, it is apparent from the model alone that both input signals are being multiplied by the same constant value. The style shown in Figure 4 does not inherently have this advantage. In this case, the variable name used in the gain blocks is of sufficient length that it does not appear in the mask of the Gain block. This can be remedied by enlarging the gain blocks, however this could have other stylistic effects. That said, if exported to a Simulink Web view, the full variable name would be visible via the rollover feature. However, the actual value of that variable would not be apparent. Thus, it can be made clear that the two gain blocks make use of the same value, but the actual value is not directly available. The availability of model workspaces if not used in a deterministic way can also cause conflicting names, which can confuse this issue. However, the most significant tradeoff between these approaches is realized when generating code.

The model in Figure 3 generates the following C function (note that the model name is `NumericalConstantProduct.mdl`).

```

void NumericalConstantProduct_step(void)
{
NumericalConstantProduct_Y.OUTPUT_1 = NumericalConstantProduct_U.INPUT_1 * 2.0;
NumericalConstantProduct_Y.OUTPUT_2 = 2.0 * NumericalConstantProduct_U.INPUT_2;
}

```

Note how the parameter defined in the Constant block appears as the numerical value 2.0 in this example. This is the same behavior as in the example in the previous section where numerical parameters were used with the *Inline parameters* option enabled. In this case, however, the numerical value is routed via a signal. Control over how this is treated during code generation is accomplished through the *Inline invariant signals* option. Clearly the signals sourced at the Constant block will not change during the simulation. Thus they are considered invariant signals. The *Inline invariant signals* option behaves for signals sourced from non-time-varying blocks exactly the same as the *Inline parameters* option behaves for parameters. This particular style also eliminates the capability to tune parameters in the generated code. Just as with the previous example and the *Inline parameters* option, the numerical representation of the data in the generated code is not suitable for making changes to these values in the software environment without regenerating code and recompiling the software project. This may not be desirable behavior in all cases.

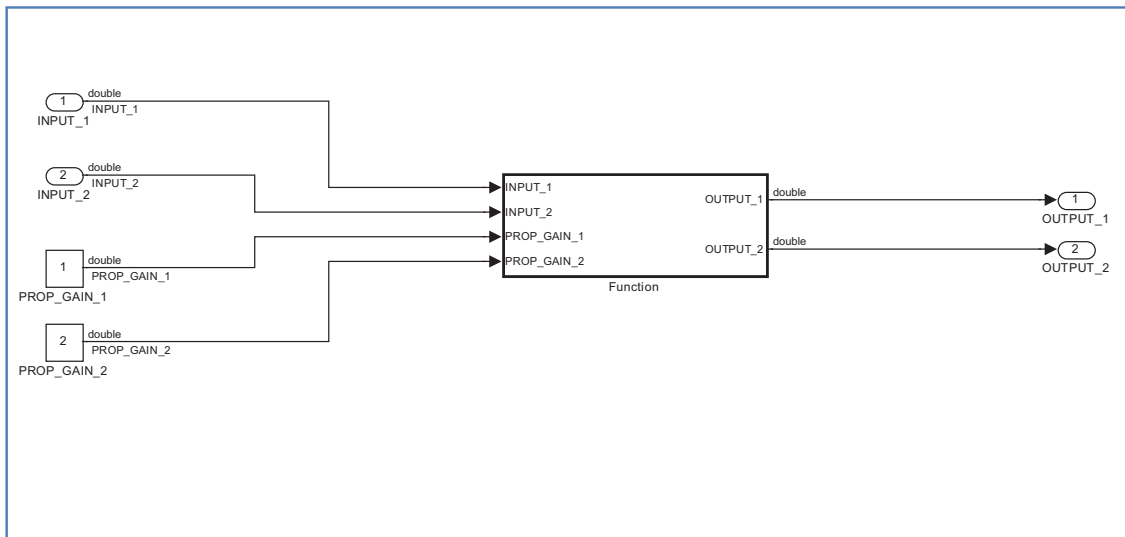
## Grouping Signals

Clearly the style in Figure 3 requires many more signal lines in a model. As models increase in size, managing signals becomes a common challenge. This is especially true when trying to ensure readability in such a model. One way to approach this problem is to group signals that have common destinations into a single signal line. This can be accomplished in one of three ways: with matrices, muxes, or buses. In order to decide which is most appropriate for a given situation, an understanding of each is necessary.

The following examples show the differences in the generated code when using individual signals, muxes, virtual buses, and non-virtual buses. Figures 5, 6, 7, and 8 show a top-level model consisting of the same function, altering only the way the interface of that function is modeled.

### Scalar, Vector, and Matrix Signals

A signal line can be used to represent scalar data, vector data, and matrix data. With scalar data, the line represents a single value. Vector and matrix data provide a multidimensional representation of numerical data. Figure 5 below shows a model with all inputs fed as individual signals.



**Figure 5: Function with individual signal I/O**

Individual signals are by far the most flexible option for defining interfaces, as they provide the most direct way to route a signal to any location in the model where it is necessary, regardless of that signal's relationship to any other signal. However, for larger models, individual signals can impede readability. Using individual signals in models with hundreds or thousands of signals can make it impossible to follow the modeled data dependencies.

The way signals are grouped, if at all, in a model will also affect the generated code. The default behavior in Real-Time Workshop Embedded Coder is to group root-level I/O into a global structure. When generating code for this function (right-click→Build subsystem), the result is as follows.

Function.h:

```

24 typedef struct {
25     real_T INPUT_1;
26     real_T INPUT_2;
27     real_T PROP_GAIN_1;
28     real_T PROP_GAIN_2;
29 } ExternalInputs;
30
31 typedef struct {
32     real_T OUTPUT_1;
33     real_T OUTPUT_2;
34 } ExternalOutputs;

```

Function.c:

```
4 void Function0_step(ExternalInputs *U, ExternalOutputs *Y)
5 {
6     Y->OUTPUT_1 = U->INPUT_1 * U->PROP_GAIN_1;
7     Y->OUTPUT_2 = U->INPUT_2 * U->PROP_GAIN_2;
8 }
```

Note that, although the signals were passed as individual arguments in the model, they are grouped in a structure in the generated code and then called from a structure. This is inconsistent with the idea of the model architecture mimicking the software architecture.

Recalling the fundamental principles for a good model architecture, the model architecture and the software architecture should match. That is accomplished in this example in the following way. If the model truly reflects the design (including the interface), then the model in Figure 5 specifies that the I/O of this function should pass individual arguments.

This behavior is controlled by the option *Pass root-level I/O* on the Interface pane of the Real-Time Workshop Configuration Parameters settings. This option is only available if the *Generate reusable code* option is enabled. The default option is *Structure reference*, whose behavior is seen in the preceding code. For consistency, in this example it is better to use the *Individual arguments* option, which passes data into and out of the function as individual variables via the function interface. This results in the following code:

```
4 void Function0_step(real_T U_INPUT_1, real_T U_INPUT_2, real_T U_PROP_GAIN_1,
5                   real_T U_PROP_GAIN_2, real_T *Y_OUTPUT_1, real_T *Y_OUTPUT_2)
6 {
7     (*Y_OUTPUT_1) = U_INPUT_1 * U_PROP_GAIN_1;
8     (*Y_OUTPUT_2) = U_INPUT_2 * U_PROP_GAIN_2;
9 }
```

Note how in this case the root-level I/O is passed via the function arguments as individual signals. This is a much more intuitive and accurate reflection of the design as shown in the model. That said, if the preceding code using the global structures was the desired output, then rather than changing the code generation options, the better solution is to alter the model. In this case, the model should depict that fact that interface data will be grouped in some way.

### Grouping Signals: Muxes

A common point of confusion arises with respect to mux and vector signals in Simulink. Strictly speaking, a true mathematical vector is best represented in Simulink as a  $[n \times 1]$  or  $[1 \times n]$  matrix. This is then mathematically compatible with all matrix math available in Simulink. As opposed to a mathematical vector, a mux signal is an array, and should be thought of like an array in C. The predominant characteristic of an array is that it is typically a collection of scalar signals of the same data type and sample time, which are then indexed numerically. Just as a C array would not contain data of different data types and or sample times, mux signals in Simulink support only the grouping of signals with the same sample time and data type.

Figure 6 shows the same model as Figure 5, however in this case the input signals have been muxed together.

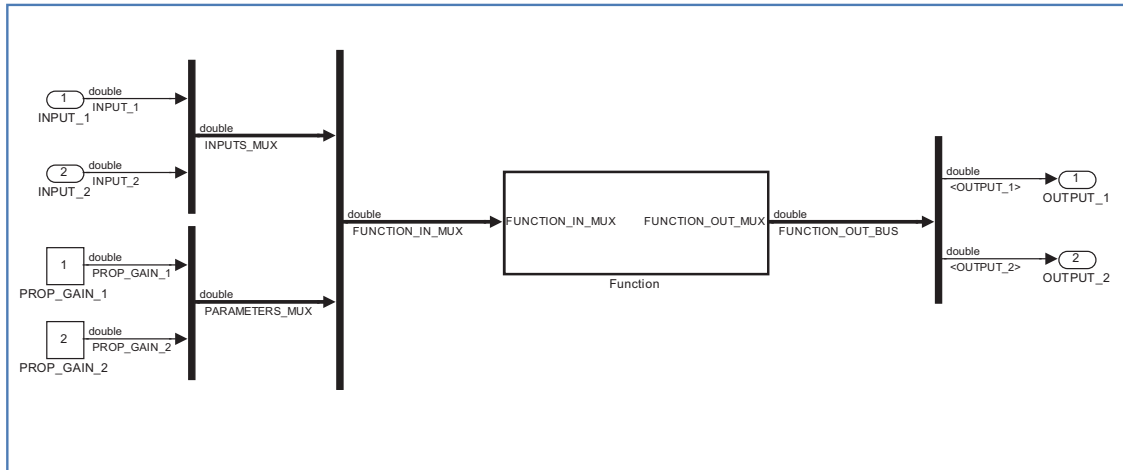


Figure 6: Function with mux signal I/O

Grouping signals into muxes can help improve readability by reducing the number of signals and potentially simplifying interfaces. This simplification, however, has no impact on modularity compared to individual signals. Muxes do significantly affect the flexibility of signal data in the model. Muxes are not a flexible method for grouping signals. Muxes by definition only allow signals of common sample time and data type to be grouped together. Also, if the structure of the mux is altered, the designer must ensure that all indexes into the mux are updated properly.

As always, it is necessary to look at the impact of mux signals on the generated code to fully understand the effect of this particular modeling construct. In this case the signals have been grouped according to inputs and parameters, hierarchically within the input mux. Generating code for the function subsystems results in the following:

```

4 void Function0_step(real_T U_FUNCTION_IN_MUX[4], real_T Y_FUNCTION_OUT_MUX[2])
5 {
6     Y_FUNCTION_OUT_MUX[0] = U_FUNCTION_IN_MUX[0] * U_FUNCTION_IN_MUX[2];
7     Y_FUNCTION_OUT_MUX[1] = U_FUNCTION_IN_MUX[1] * U_FUNCTION_IN_MUX[3];
8 }

```

The relationship between the mux signals in the model and the code implementation is clear. The mux signals are treated as arrays in C. Note how the array in the generated code is declared of type `real_T`, meaning that all of the individual signals in the array are of type `real_T`. This illustrates the parallel with muxes in Simulink, and why they do not support mixed data types.

It is also important to note that the hierarchy of the muxed signals in the model is completely lost in the generated code. This is not surprising, as arrays are not a hierarchical data type in C. Thus, a best practice shown in this example is to avoid hierarchical mux signals in the model, as that hierarchy has no functional meaning. If the code seen above is the desired output, then the model should be altered to eliminate the hierarchical muxes, so that the model is an accurate description of the function interface in the software environment. If the model is accurate, and the hierarchical interface definition is important, then a mux signal is not the proper choice and the model should be altered to make use of a signal type that allows for hierarchy. Bus signals are the appropriate way to accomplish such a task.

### Grouping Signals: Busses

A bus signal is a grouping of signals purely for organization. As such, it is not subject to the same constraints as a mux signal and can group signals that are of different types and different

sample times. That said, perhaps the most fundamental difference between mux and bus implementations is that accessing an individual signal from a mux must be done using a numerical index. Accessing a signal from a bus, however, is done by referencing the name of that signal.

### Virtual vs. Non-Virtual Busses

Recall from the model architecture discussion<sup>(1)</sup> that subsystem hierarchies can have varying levels of functional meaning. On one extreme is the virtual subsystem, which is merely a graphical construct for organization, with no functional meaning. The other extreme is a referenced model, where the boundary is not only functional, but requires a deterministic definition of the interface. Virtual and non-virtual busses provide this same spectrum of functional meaning as the varying kinds of subsystems. In fact, exploiting the parallelism between hierarchical groupings of signals and blocks is an extremely powerful stylistic approach.

A virtual bus is the analogous construct for signals to a virtual subsystem. A virtual bus allows a hierarchical organization of the signals in a model, however the virtual bus has no functional meaning in either the model or the generated code. Like a virtual subsystem, it is simply for graphical organization and readability. Figure 7 shows the same function as Figures 5 and 6, but instead of individual ports or mux signals, the interface uses a virtual bus input and output.

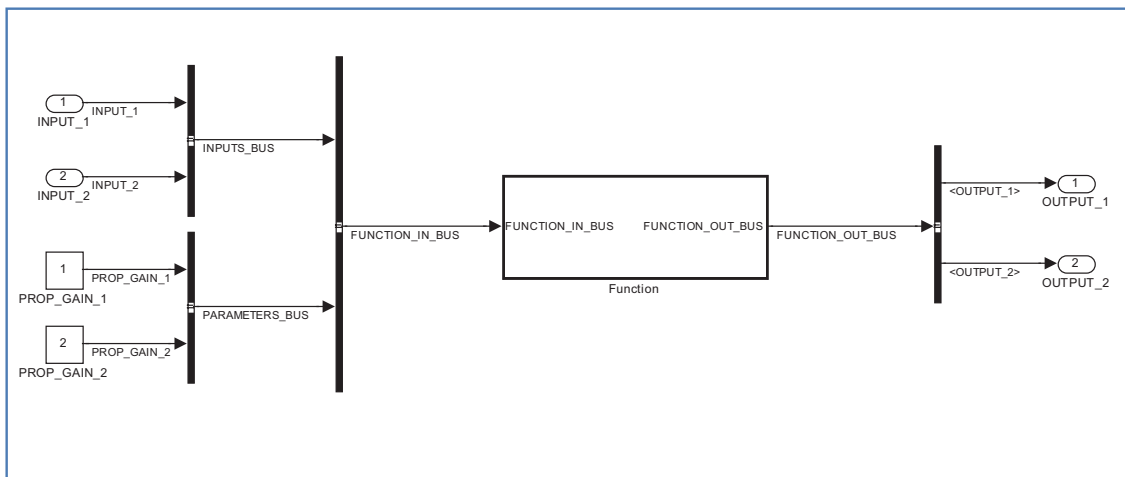


Figure 7: Function with virtual bus I/O

When considering readability, the ability to access signals via name is perfectly consistent with the idea of the model as a self-contained, fully-specified document. A good model architecture makes use of the virtues of using a hierarchical subsystem architecture for organization.<sup>(1)</sup> The same hierarchical organization for signals can be accomplished with bus signals, as they allow a hierarchical construction (a bus signal within a bus signal). More simply, bus signals provide the hierarchical organization for signals that subsystems provide for blocks.

The one caveat is that when large busses are crossing partitions within a model, it can become unclear which signals are actually being used. Consider a Simulink “Air Data” bus crossing a model reference boundary which represents a portion of the control system logic. Passing the bus in its entirety is inefficient as it does not explicitly specify which signals that particular portion of the algorithm requires. From the standpoint of accurately modeling data dependencies and

function interfaces, this is undesirable. One method to avoid this issue is to develop a modeling style which explicitly selects the required signals before passing them into the model reference. This allows the interface bus to exactly model the function interface, providing a more accurate and readable description of the function both in the model and the code. However, this sub-selection of bus elements to create the desired interface bus can result in extraneous data copies in the generated code. The optimal balance between these two concepts becomes a question of model and software architecture.

The effect of bus usage on modularity depends on whether the busses are virtual or non-virtual. For virtual busses, there is no impact on modularity. Virtual busses are also perhaps the most flexible construct for grouping and organizing signals. They do not require the numeric indexing of muxes. They also support hierarchical organization of signals. And virtual busses do not require bus objects, meaning they are not dependent on such definitions in the base workspace.

Whether a bus is virtual or non-virtual has important consequences on code generation. The following examines the treatment of virtual busses. With the *Pass root-level I/O* option set to Structure Reference, the generated code from the model using virtual busses in Figure 7 appears as:

Function.h:

```

24 typedef struct {
25     real_T INPUT_1;
26     real_T INPUT_2;
27     real_T PROP_GAIN_1;
28     real_T PROP_GAIN_2;
29 } ExternalInputs;
30
31 typedef struct {
32     real_T OUTPUT_1;
33     real_T OUTPUT_2;
34 } ExternalOutputs;

```

Function.c:

```

4 void Function0_step(ExternalInputs *U, ExternalOutputs *Y)
5 {
6     Y->OUTPUT_1 = U->INPUT_1 * U->PROP_GAIN_1;
7     Y->OUTPUT_2 = U->INPUT_2 * U->PROP_GAIN_2;
8 }

```

Note that this is exactly the same as the original code generated from the model in Figure 5 that used individual scalar input signals. This accentuates the importance of a modeling standard that prevents such behavior. In this case, because the model groups signals into a single data element (the `FUNCTION_IN_BUS/FUNCTION_OUT_BUS` signals), it is intuitive that in the generated code these signals also be grouped into a single data element (the `ExternalInputs/ExternalOutputs` structures).

However, there is still an inconsistency between the model and the generated code for Figure 7. Note that in the model, the input bus is hierarchical, containing underneath it `INPUTS_BUS` and `PARAMETERS_BUS`. However, these busses do not appear anywhere in the generated code. Hierarchical data types were previously stated to be one of the key differences between muxes and busses. So why is there no hierarchy? The reason for this is that the model in Figure 7 makes use of virtual bus signals.

Recall from above that virtual busses are best thought of like virtual subsystems, in that they are a purely graphical construct to organize signals in a model. Like virtual subsystems, virtual busses have no functional meaning. As such, during code generation the virtual bus is flattened and the elements of a virtual bus are treated as individual arguments. This is in fact why the exact

same code can be generated from the model in Figure 7 and the model in Figure 5: the two models are functionally equivalent. Giving signal groupings a functional meaning is accomplished with non-virtual bus signals and bus objects.

Figure 8 is the same as Figure 7, however the interface uses a non-virtual bus input and output. If a virtual bus is analogous to a virtual subsystem, a non-virtual bus is the analogous construct to a model reference. A non-virtual bus requires the definition of a bus object. A bus object has the same use for a bus that a data object has for a parameter or signal. The bus object specifies sample times, data types, and sizes of all of the signals within the bus. Aside from the additional level of specification that a bus object defines, it also has an effect on the generated code. A non-virtual bus with a bus object definition will appear as a structure in the generated code.

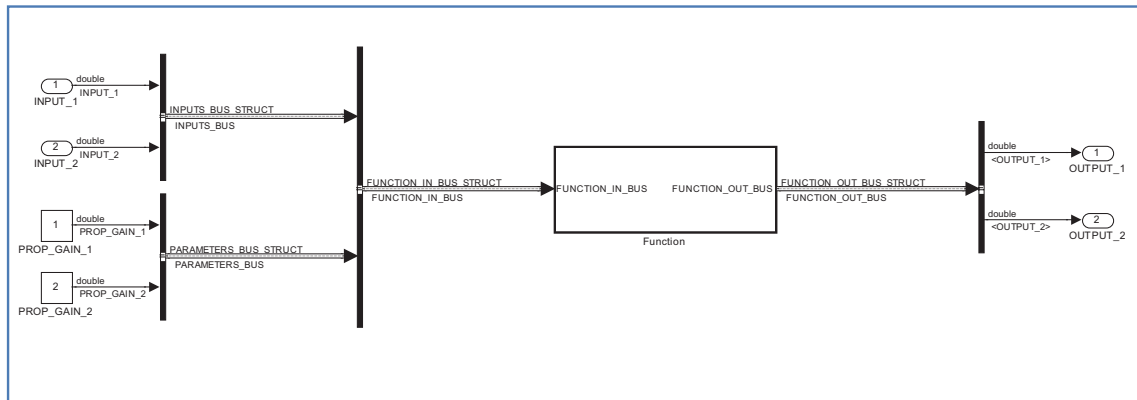


Figure 8: Function with non-virtual bus I/O

Other than a slight difference in the appearance of the signal lines in the model, virtual and non-virtual buses are equivalent in terms of readability. However, though not explicitly apparent in the model, non-virtual busses require a `Simulink.Bus` object be defined in the base workspace. This dependency means that the files containing the bus definition data (MAT files, .m files, etc.) must be managed separately and kept synchronized with the proper version of the model. This clearly affects the modularity of the model.

Non-virtual busses are purposely inflexible. When the bus structure or interface across model reference is changed, the corresponding bus object must be changed. If the bus objects are loaded from or created by a file, that file must be updated as well. Many users have adopted a workflow that employs both virtual and non-virtual busses. Although clearly case dependent, this is one way to achieve the optimal balance for defining function interfaces.<sup>(3)</sup>

The difference between virtual and non-virtual buses is best understood by examining the code generated from the model in Figure 8 using non-virtual busses. Recall that the use of model reference requires bus object definitions for root-level I/O. In the example above, generating code with the default options from the referenced model `Function.mdl` would look as follows:

Function\_types.h:

```

14 #ifndef _DEFINED_TYPEDEF_FOR_PARAMETERS_BUS_STRUCT_
15 #define _DEFINED_TYPEDEF_FOR_PARAMETERS_BUS_STRUCT_
16
17 typedef struct {
18     real_T PROP_GAIN_1;
19     real_T PROP_GAIN_2;
20 } PARAMETERS_BUS_STRUCT;
21
22 #endif

```

```

23
24 #ifndef _DEFINED_TYPEDEF_FOR_FUNCTION_IN_BUS_STRUCT_
25 #define _DEFINED_TYPEDEF_FOR_FUNCTION_IN_BUS_STRUCT_
26
27 typedef struct {
28     INPUTS_BUS_STRUCT INPUTS_BUS;
29     PARAMETERS_BUS_STRUCT PARAMETERS_BUS;
30 } FUNCTION_IN_BUS_STRUCT;
31
32 #endif

```

### Function.c

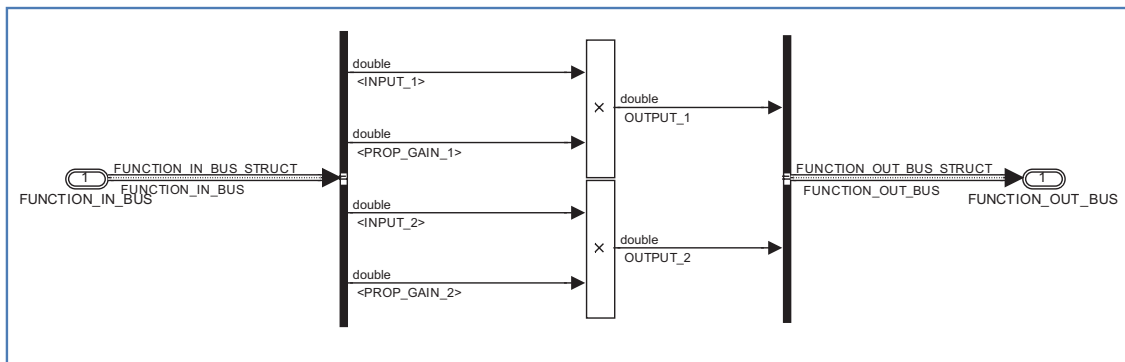
```

21 void Function0_step(FUNCTION_IN_BUS_STRUCT *U_FUNCTION_IN_BUS,
22                   FUNCTION_OUT_BUS_STRUCT *Y_FUNCTION_OUT_BUS)
23 {
24     (*Y_FUNCTION_OUT_BUS).OUTPUT_1 = (*U_FUNCTION_IN_BUS).INPUTS_BUS.INPUT_1 *
25     (*U_FUNCTION_IN_BUS).PARAMETERS_BUS.PROP_GAIN_1;
26     (*Y_FUNCTION_OUT_BUS).OUTPUT_2 = (*U_FUNCTION_IN_BUS).INPUTS_BUS.INPUT_2 *
27     (*U_FUNCTION_IN_BUS).PARAMETERS_BUS.PROP_GAIN_2;
28 }

```

As opposed to the previous examples, this version of the code makes use of C-structures. Furthermore, these structures are defined exactly according to the bus objects defined in the model. This allows the use of hierarchical buses in the model, where the generated code properly reflects such a design decision by using embedded structures in C. This relationship between bus objects and C structures is the key to a successful modeling style that uses non-virtual buses.

It is important to remember that bus objects are exactly like C structures, in that they are a typedef. This is also true in Simulink, where a signal defined by a bus object must be thought of as a signal of that type, just like a double, single, etc. In fact, examining the model `Function.mdl` shows this exactly.



**Figure 9: Inside Function.mdl**

In this view, the option to display port data types is enabled. Note that the signal outputs of the bus selector blocks all have data type double. Furthermore, note that the bus signals have type `FUNCTION_IN_BUS_STRUCT`/`FUNCTION_OUT_BUS_STRUCT` respectively. This exactly mimics the behavior of structures in C. Keeping this in mind makes working with non-virtual buses in Simulink much less confusing.

### Signals Summary

As models become larger and larger, the need arises to organize signals for readability and signal data management. The primary constructs for doing so in Simulink are muxes, virtual busses, and non-virtual busses. As discussed, each has benefits and drawbacks in terms of

readability, modularity, flexibility, and code generation. A signal management style must be selected and enforced that meets the needs of the application and that will generate suitable code.

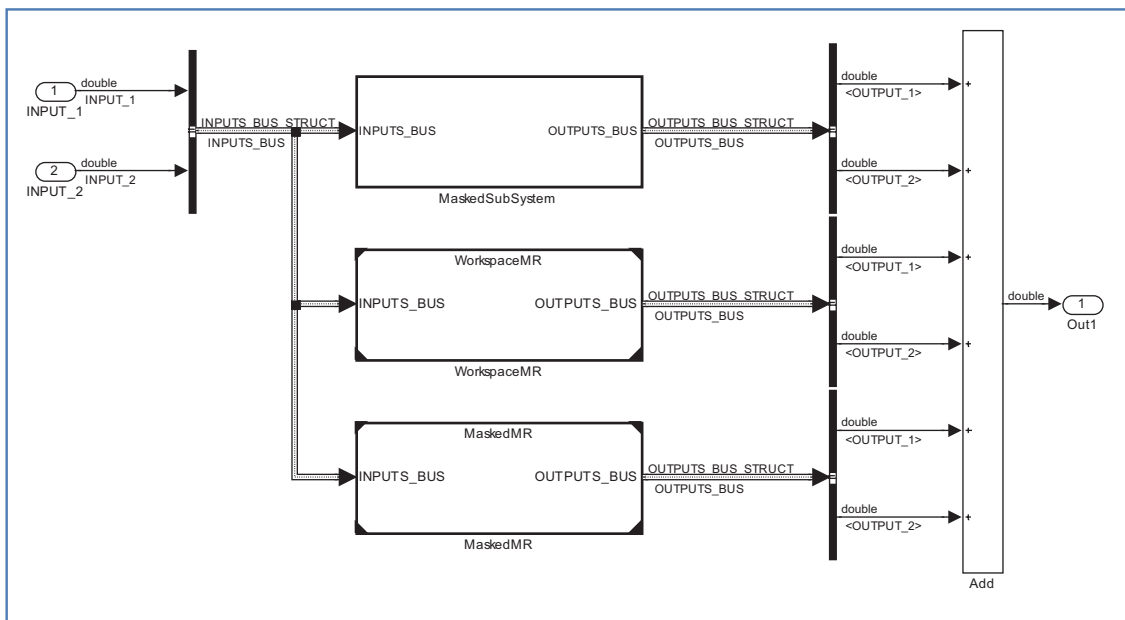
## COMBINING SIGNALS AND PARAMETERS

Having a clear understanding of the implementations of parameters and signals, the next question is when to use which implementation and how to combine them. As always, many of these considerations come from thinking of the code implementation of the algorithm. Earlier in this discussion, we made the distinction that signals vary with time and parameters do not. While that is a good rule of thumb, there are often multiple ways to accomplish the same task in Simulink. This flexibility makes the tool very extensible, but it is important to understand the goal of the models and develop a style that supports that goal.

It should be noted that in the preceding examples, the parameters were being passed via signal lines into the models, and thus were automatically included as part of the function interface. While these signals do not vary with time, this is a perfectly acceptable modeling style that actually promotes readability.

Another option for passing parameters is using workspace variables passed through block masks. Figure 10 shows three different methods for accomplishing this:

1. Masked atomic subsystem with parameters defined in the base workspace. Subsystem parameters set to generate a reusable function.
2. Model reference with parameters defined in the model workspace.
3. Masked model reference with parameters defined in the base workspace.



**Figure 10: Three different function/parameter interfaces**

Code generated at the top level of this model appears in the following example, which shows the different function interfaces resulting from the different parameter implementations.

## InterfaceEx.h

```
46 #define PROP_GAIN_1 1.0
47 #define PROP_GAIN_2 2.0
```

## InterfaceEx.c:

```
50 InterfaceEx_MaskedSubSystem(&rtb_INPUTS_BUS, &B.MaskedSubSystem, PROP_GAIN_1,
51 PROP_GAIN_2);
52
53 /* end of Outputs for SubSystem: '<Root>/MaskedSubSystem' */
54
55 /* OutputUpdate for ModelReference Block: '<Root>/WorkspaceMR' */
56 mr_WorkspaceMR(&rtb_INPUTS_BUS, &rtb_OUTPUTS_BUS);
57
58 /* OutputUpdate for ModelReference Block: '<Root>/MaskedMR' */
59 mr_MaskedMR(&rtb_INPUTS_BUS, &rtb_OUTPUTS_BUS_p, PROP_GAIN_1, PROP_GAIN_2);
```

Note how in each of these three cases, a function call was generated with that function defined in a separate file. Also note how the `OUTPUTS_BUS` name is mangled. This is because the same signal name was used for the output of all three instances. Clearly this is a non-optimal modeling style decision. This was done however, so that the signal interfaces were exactly the same for each instance, making the differences due to the parameter implementations the only difference.

Examining the function `InterfaceEx_MaskedSubSystem` shows that in this case the parameters (named `PROP_GAIN_1` and `PROP_GAIN_2`) are passed into the function as individual arguments. The benefit of this approach is that the parameters can then be declared in whatever form was specified in the model. In this case, as seen in `Function.h`, the parameters were defined using `mpt.Parameter` objects with a selected storage class of `#define`.

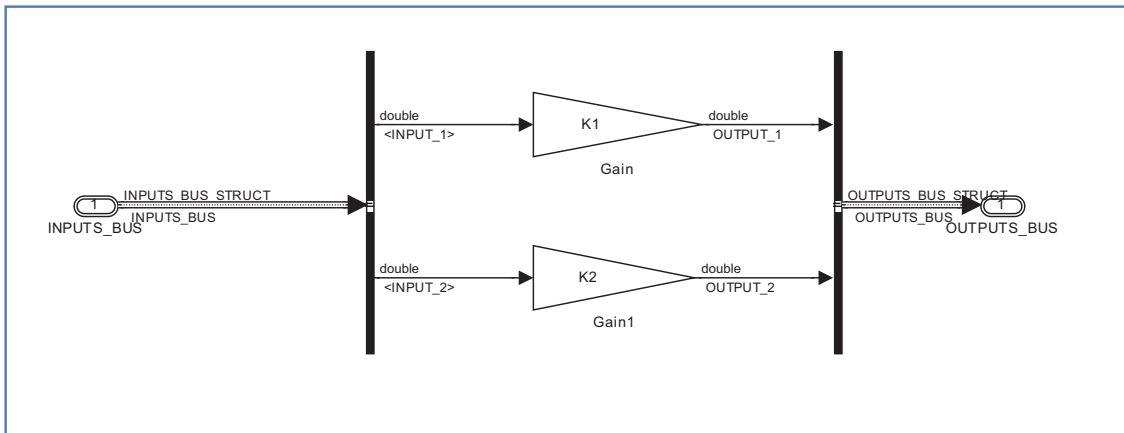


Figure 11: Looking under the masked subsystem

Figure 11 shows the algorithm underneath the masked subsystem in Figure 10. Note that because of the mask, the variables in the gain blocks are not `PROP_GAIN_1` and `PROP_GAIN_2`, but rather `K1` and `K2` respectively. If we examine the generated code, we should see the code parallel of this.

## InterfaceEx\_MaskedSubSystem.c

```
25 void InterfaceEx_MaskedSubSystem(const INPUTS_BUS_STRUCT *rtu_INPUTS_BUS,
26 rtB_MaskedSubSystem_Figure12 *localB, real_T rtp_K1, real_T rtp_K2)
27 {
28 /* BusCreator: '<S1>/Bus Creator' incorporates:
29 * Gain: '<S1>/Gain'
30 * Gain: '<S1>/Gain1'
```

```

31     */
32     localB->OUTPUTS_BUS.OUTPUT_1 = rtp_K1 * (*rtu_INPUTS_BUS).INPUT_1;
33     localB->OUTPUTS_BUS.OUTPUT_2 = rtp_K2 * (*rtu_INPUTS_BUS).INPUT_2;
34 }

```

This is one benefit of using masks. They allow multiple instances of a block with different parameters values. In fact, this is directly related to the usefulness of libraries. Imagine this masked subsystem implemented in a library. Because the mask has its own workspace (where  $K_1$  and  $K_2$  are stored), multiple instances of this block could exist in a model without changing the block itself. However, each instance could operate with different parameters by passing different values through the mask. Essentially, then, the mask workspace can be thought of as local data for the function. The mask simply exposes that local data to the function interface. The second example is based exactly on this idea of local data.

Now consider the second method whereby the functionality in Figure 11 is included as a Model Reference block. In this case, rather than implementing it as a subsystem, the functionality is implemented as a referenced model. Like the mask workspace, each model has its own workspace, called the model workspace (as described in the previous section on Defining Parameters via Data Objects). In this instance, the parameters are defined in the model workspace rather than the base workspace.

Recall that this instance generated the following function interface:

```

55     /* OutputUpdate for ModelReference Block: '<Root>/WorkspaceMR' */
56     mr_WorkspaceMR(&rtb_INPUTS_BUS, &rtb_OUTPUTS_BUS);

```

Note that the parameter values are not part of the function interface. So where are these parameters defined? Following the thought process of treating the model workspace like local data, the answer is found by inspecting the code for the function `mr_WorkspaceMR`.

```

4     void mr_WorkspaceMR(const INPUTS_BUS_STRUCT *rtu_INPUTS_BUS, OUTPUTS_BUS_STRUCT *
5                          rty_OUTPUTS_BUS)
6     {
7         (*rty_OUTPUTS_BUS).OUTPUT_1 = 1.0 * (*rtu_INPUTS_BUS).INPUT_1;
8         (*rty_OUTPUTS_BUS).OUTPUT_2 = 2.0 * (*rtu_INPUTS_BUS).INPUT_2;
9     }

```

Note that in this case, the parameters are implemented as numerical values in the code (1.0 and 2.0 respectively). This is exactly as expected. The parameters in this case are used as local data in the generated code. That said, recall the previous discussion on tunability of parameters in the generated code. Clearly, this implementation does not support this idea, as the concepts of locally scoped data and tunability tend to conflict.

However, also recall that for high integrity applications, scoped data is far more deterministic and modular, making the model workspace a very valuable tool. However, in the instance where tunability of the generated code is required, the question then becomes, is there a way to have tunable parameters when using a referenced model? One solution was seen in the previous section, where the parameters were implemented as signals and passed via the model reference interface. The second option is to define the parameters in the base workspace and pass them as arguments to the model reference mask. This is very similar to the method used with the masked subsystem, however passing parameters to a referenced model is a bit more complicated. When masking a subsystem, the mask creates its own workspace and variables in that workspace are automatically scoped properly so that blocks can see those values. Recall the fundamental difference between atomic subsystems and referenced models. Referenced models require a hard definition of the interface. This includes the parameter arguments. Therefore, parameter

arguments must be explicitly defined in order to pass them to the referenced model. This is done using the model workspace as in the previous example.

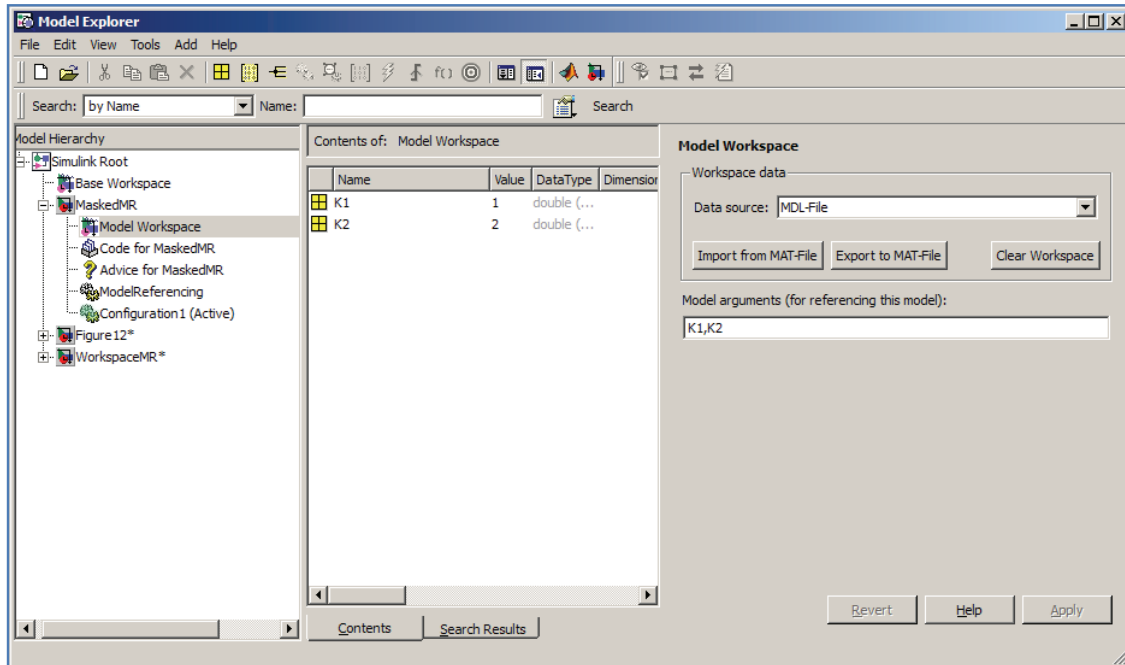


Figure 12: Model Explorer showing definition of model reference parameter arguments

Note that in this case, the variables K1, K2 are listed in the right-most pane, in the field *Model arguments (for referencing this model)*. Unlike just using the model workspace as before, the *Model arguments* define the data that is required by the referenced model, and specifies that it need be supplied through the model's mask. In doing so, the model reference parameters mask appears as:

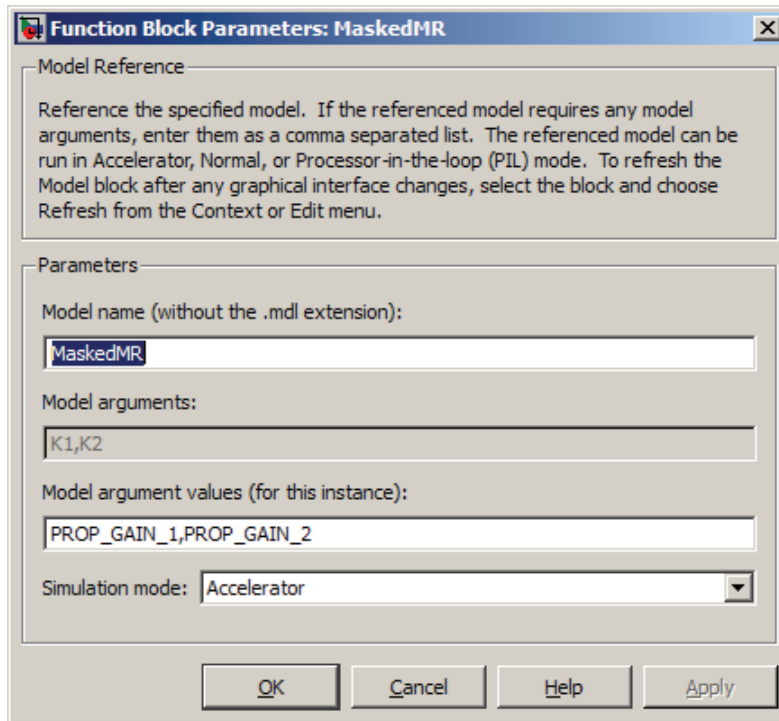


Figure 13: Model reference parameters mask

Note that in this mask, the *Model arguments* field displays the parameters that were defined as part of the model interface. Also note that these are grayed out, meaning that the interface is a hard definition, and only the parameters needed by the function can be passed via this interface.

Having specified these parameters as inputs, the *Model argument values (for this instance)* field then allows the user to pass in values from the base workspace. This functions exactly the same as the masked subsystem. In this example, the values `PROP_GAIN_1` and `PROP_GAIN_2` are specified in the base workspace, and are exactly the same as those passed into the masked subsystem. Recall the function interface for the masked model reference appeared as:

```
59   mr_MaskedMR(&rtb_INPUTS_BUS, &rtb_OUTPUTS_BUS_p, PROP_GAIN_1, PROP_GAIN_2);
```

Note the inclusion this time of the parameters in the function interface. Examining the code for the function `mr_MaskedMR`

`Mr_MaskedMR.c`

```
4   void mr_MaskedMR(const INPUTS_BUS_STRUCT *rtu_INPUTS_BUS, OUTPUTS_BUS_STRUCT
5                       *rty_OUTPUTS_BUS, real_T rtp_K1, real_T rtp_K2)
6   {
7       (*rty_OUTPUTS_BUS).OUTPUT_1 = rtp_K1 * (*rtu_INPUTS_BUS).INPUT_1;
8       (*rty_OUTPUTS_BUS).OUTPUT_2 = rtp_K2 * (*rtu_INPUTS_BUS).INPUT_2;
9   }
```

Note that this is equivalent to the code in `InterfaceEx_MaskedSubSystem.c`. This raises the question of which method should be chosen.

The answer lies within the polymorphic nature of libraries. A model reference implementation of this same functionality for different data-type inputs would require three separate and distinct models in Simulink that include the three different interfaces. This would then also generate three individual function calls. The model and the generated code would be more parallel. However, this requires explicitly managing the three separate models. This is part of the balance that must be struck when determining the appropriate places for library and model reference usage. In this example, the library function in question is a simple function that will be used many times and will not be updated often. In cases like this, a library may be best choice.

## CONCLUSION

This paper has covered the basics of defining data in a Simulink model via the two most fundamental data types: signals and parameters. The discussion reviewed the basic methods used to define signals and parameters and the related impact on modularity, readability, flexibility, and code generation. When developing a modeling style, these are the primary factors that should be considered.

There is no one single modeling style that is “best” for all situations. Instead, the thought process is what is most important. As with choosing a model architecture, working with the data for a model is best done given an understanding of the expected use and output of the model. This is especially true if the model is intended for production code generation. In these cases, it is very important to work with the software engineers early to ensure that the modeling style reflects the desired coding style. It is far easier and less costly to gain this understanding early in the process rather than change the modeling style later. Specifically, when using Model-Based Design for the development of high-integrity software, any data required for the model should be implemented in a way that parallels the desired implementation of that data in the generated code.

## REFERENCES

1. *Model-Based Design for Large Safety-Critical Systems: A Discussion on Model Architecture.* **Anthony, Mike and Friedman, Jon.** San Diego, CA : s.n., 2008. AUVSI Unmanned Systems.
2. *Configuration Management of the Model-Based Design Process.* **Walker, Gavin Friedman, Jonathan, and Aberg, Rob.** Society of Automotive Engineers. SAE 2007 TRANSACTIONS. JOURNAL OF PASSENGER CAR: ELECTRONIC AND ELECTRICAL SYSTEMS Section 7 - Volume 116
3. *Best Practices for Managing Bus Data in Simulink®.* MathWorks Note, Tech. <http://www.mathworks.com/support/tech-notes/1800/1822.html>