2009-01-0155

Methods for Interfacing Common Utility Services in Simulink Models Used for Production Code Generation

Jeffrey M. Thate and Robert A. Kagy
Caterpillar, Inc.

Robyn A. Jackey, Roger Theyyunni, Jagadish Gattu
The MathWorks, Inc.

Copyright © 2009 SAE International

ABSTRACT

Traditionally, code generated from Simulink models has been incorporated into production applications in a manner similar to hand-written code. As the size of the content created in Simulink has grown, so has the desire to do more integration in Simulink. Integrating content from C/C++ calling environments directly into Simulink blocks rather than just calling external legacy code prevents errors and preserves signal flow visibility in the Simulink models.

Although much of the application content has transitioned to Simulink models, most of the Common Utility Services (e.g., communications, diagnostics, and nonvolatile memory) still exist in C/C++ libraries. While application content changes frequently, Common Utility Service content changes infrequently and is heavily leveraged across many applications. Therefore, it is often desirable to call these Common Utility Services from their existing C/C++ libraries rather than porting them to be generated directly from Simulink models.

Many common services do not fit easily into a constant parameter and dynamic signal flow approach that is typical of Simulink models. This paper examines methods used for creating custom blocks and non-graphically represented code to create a Simulink interface to these Common Utility Services.

INTRODUCTION

Traditionally, code generated from Simulink models has been incorporated into production applications in a manner similar to hand-written code [1]. However, as the size of the Simulink content has grown, so has the desire to do more integration in Simulink; thus preventing errors and the loss of signal flow visibility, which occur from transitioning from Simulink models to C/C++ calling environments. Although much of the application content has transitioned to Simulink models, most of the Common Utility Services still exist in C/C++ libraries. This paper will examine what constitutes Common Utility Services, the motivation behind integrating them more tightly in Simulink, and the typical methods used to interface between Simulink models and Common Utility Services. Finally, this paper will conclude by identifying various obstacles to incorporating Common Utility Services in Simulink.

COMMON UTILITY SERVICES

WHAT ARE COMMON UTILITY SERVICES? - Common Utility Services provide application-independent interfaces to ECU platform functions. These interfaces include functions such as communications, diagnostics, and the use of nonvolatile memory. This Common Utility Service content changes infrequently and is heavily leveraged across many applications. Common Utility Services often constitute a large percentage of the overall application, but a small percentage of the content specific to that particular application.

Common Utility Service content has historically been created in environments outside of Simulink. The use of Simulink in embedded systems is a relatively recent phenomenon. Most experts writing these utilities come from a software background and are typically more fluent in traditional software languages.

THE NEED FOR COMMON UTILITY SERVICES INTEGRATION - Simulink has become a common platform for specifying application control content. As the bulk of the content that is changing between applications is being described in Simulink, the interface between Simulink and Common Utility Services is increasing. Because a majority of the application-specific content is expressed in Simulink, it is natural to expose those interfaces in the location where they are being integrated.

The Requirement for Lean, Simple Interfaces - The large number of connections from Simulink applications to Common Utility Services requires lean, easy-to-use interfaces. These interfaces will be used repeatedly in many places, so each type of service should be highly

optimized to minimize overhead. Excess overhead can result in run-time failures if the service cannot run at the required rate or if the service fails to operate within the memory constraints.

If the Common Utility Service interfaces are easy to use, the interfaces can become a layer of abstraction that can be understood by a broader group of developers. Often through the use of automation, GUIs, and integrated context-sensitive help, engineers who are not familiar with a service can become productive using that service.

The interface to connect the services should require user entry of information in a single location. Duplication of data entry is a common source of errors. A common example of this problem is using global variable names that must match between hand-written utility services and their usage in Simulink models. This often occurs when using one of the simple forms of integrating Common Utility Services in which the Simulink blocks only provide a point of execution and refer to C code variables that perform the configuration of the service. In this case, the interface to the service is spread across both Simulink blocks and hand-written C code. This code is separately maintained and connected by a common reference such as a variable name. It is easier to initially implement the service in this manner because individual configurations for the service don't need to be exposed to the Simulink service block. However, it can be much more difficult to use and maintain the usages of them.

The examples shown below in Figures 1 and 2 are for a CAN receive block. Figure 1 shows the block mask that provides a point in which to specify the C code structure used to configure this service instance. Figure 2, shows an excerpt of C code that could be used to configure this instance of the block.

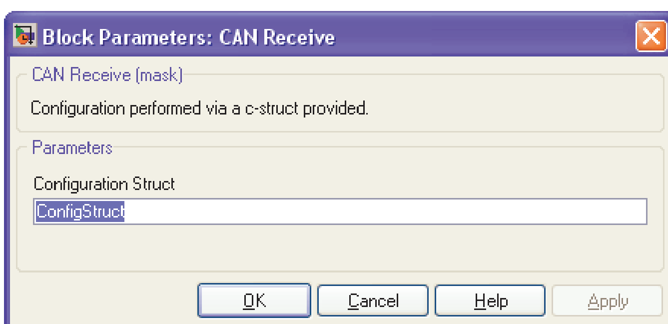

Figure 1. CAN receive block mask referencing C code configuration structure.

```
CanReceive_t ConfigStruct = {
 CAN_PORT_A,
  1, /* CAN Identifier */
  CAN_11BIT /* Identifier Type */
  …
};
```

Figure 2. CAN receive C code configuration structure.

<u>The Requirement for Centralized Implementation - Typically</u> Common Utility Services have a single point of configuration of aspects of the service that are not instance-specific. Often a setup block is provided for that service to provide a calling point for creating non-instance specific code. An example of this would be the CAN Setup block provided in the xPC Target library (Figure 3), which provides (among other things) the ability to configure the baud rate of each CAN link [2].

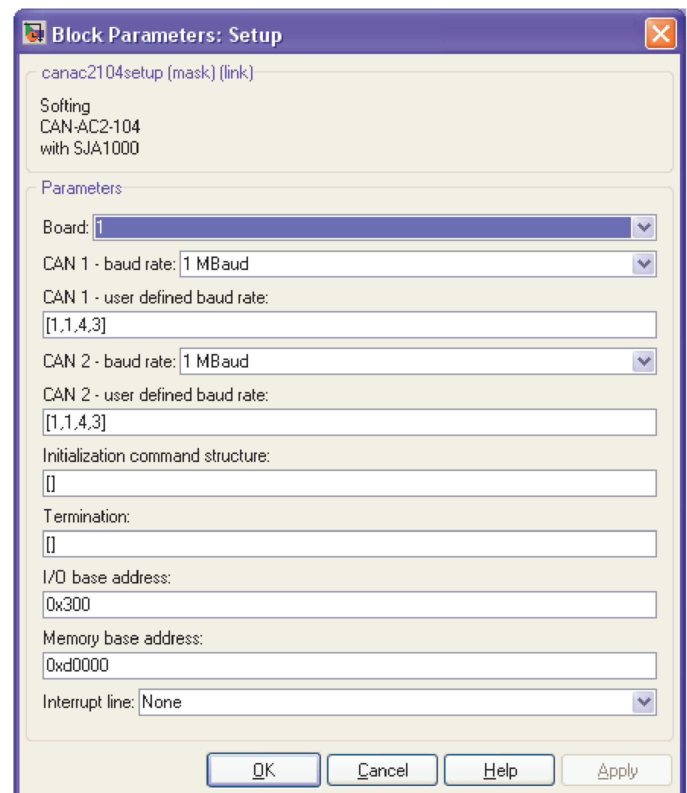

Figure 3. xPC CAN setup block mask.

Some of the configuration parameters required by the setup block may be based on instances of usage of that service. For example, a setup block configuration parameter for a CAN service might be the number of CAN messages to be transmitted.

Traditionally, C code implementations have either relied on run-time code to collect instance specific information and calculate the required configuration parameter or required the user to do collect the instance specific information and provide the calculated configuration parameter manually. Doing these calculations at run time automates the process at the expense of target resources (RAM, ROM, and execution time). Additionally, not all information may be easily available at run time. Requiring the user to track this information can reduce resource requirements at the expense of increased likelihood of errors due to the manual steps required to correctly calculate the values and the need to continually recalculate these values as usage patterns change (e.g., an additional CAN transmit was added in the above example).

In many cases, the steps required to calculate these centralized configurations can be complex. They can be based directly on instances of that service (e.g., the number of CAN transmit and receive blocks in the above example) as well as other services that use the service (e.g., a diagnostic service that accesses the C code CAN service).

Tracking the information needed to calculate these centralized configuration parameters may be difficult and costly to do at run time, and difficult and error-prone to do manually. The use of the code generator provides the best of both worlds: automated calculation often with access to information not normally easily accessible at run time, and the ability to do this calculation during code generation instead of at run time; thus, reducing target resources.

Using code generation often enables less experienced users of the service to perform at or above the level of much more experienced users who must manually track instance usage information required to calculate configuration parameters. The use of code generation empowers users to have a good understanding of the application they are developing without needing to fully understand how the underlying services work. While this may create challenges, it also brings many benefits. Increased productivity is possible through the reduction of interfacing across multiple engineering groups. Often, tradeoffs between configuration options are difficult to manually track, even for those who have a good understanding of the principles of their operation. The simple interface can improve users' understanding of a service, often resulting in better utilization of that service for a particular application.

An example illustrating the benefits of automating the configuration of Common Utility services can be seen in one aspect of a typical CAN Service. The CAN Common Utility Service requires the configuration of the number of packet objects to allocate. These packet objects are used to store received CAN packets and CAN packets waiting to be transmitted. The number of packets should be minimally set to reduce overall RAM usage. However, if the number of packets is not sufficient, the messages to be transmitted or received may be lost due to a lack of allocated memory to store the message. The actual worst-case sizing of the number of packets can be determined automatically but requires multiple pieces of information about how each CAN block is used (such as its execution rate, etc.). When this service is used in hand-written code, it is up to each individual user to ensure the service is configured correctly. Simulink provides the ability to have a system-wide view of CAN usage, and automation via the code generator that allows automatic configuration of this aspect of the service simplifying user interface and reducing possible errors due to incorrect configuration.

Another simple example based again on the previous CAN example is the automation of mask/match filters typically used in CAN hardware. These filters are used to limit the number of messages received through the CAN hardware. Typically, it is best to make the filter as restrictive as possible – thus, reducing the execution load on the ECU and possibly making it simpler for the application to determine how to route the messages received. As long as CAN identifiers are known at code generation time, it is possible for the code generator to determine the most restrictive filters that will allow receipt of the desired messages. Automating the calculation of filters prevents errors that can occur when it is left to the user to manually configure them.

Simulation Benefits - Simulation of Common Utility Services can greatly aid in the development of some applications. Some services can execute in a meaningful way during simulation, allowing more efficient development of applications using those services. For example, diagnostic systems are beginning to add models of sensors that are used to validate the diagnostic information obtained from that sensor. These sensor models generally require simulation during their development, and often benefit from being simulated in the context of the rest of the diagnostic system

A further example involving diagnostics is the movement towards using internal states of a diagnostic Common Utility Service in the application. An example is the requirement to determine if a particular diagnostic has been logged and if so, to take some particular action, such as derating the engine power. The logged state of a diagnostic is often an internal state of the diagnostic system. To fully simulate the application in this case, the diagnostic system would also have to be simulated.

Rapid Prototyping Benefits - The addition of Common Utility Services into custom targets that support production ECUs allows more prototyping to occur directly on the final intended target hardware. Traditionally, custom high-performance targets have been used to do rapid prototyping [3]. However, as code generators have become more efficient, and target hardware has become more powerful, including features such as hardware floating-point support, the need for custom rapid prototyping hardware has diminished. The inclusion of Common Utility Services directly in Simulink models provides one of the remaining pieces of functionality that is typically available only on rapid-prototyping targets.

The benefits of using the intended production hardware for rapid prototyping are numerous: lower hardware costs, earlier detection of hardware issues, better understanding of possible final production target constraints (e.g., RAM/ROM and execution time), and the use of common tools and processes throughout development. Additionally, some services may be available using this approach that is not commonly available on rapid prototyping targets (e.g., nonvolatile memory support). While we have found production targets can easily be used for most development programs, there will always remain development programs that are pushing the boundaries of what is

possible in production targets. For these projects, traditional rapid prototyping targets will remain the primary development hardware.

INTERFACING COMMON UTILITY SERVICES IN SIMULINK

MEHTODS FOR SCHEDULING COMMON UTILITY SERVICES - As described above, each Common Utility Service has been implemented by breaking the service into two general pieces, a setup block and an instance-specific feature block. Many services require passing information between blocks not represented directly in Simulink by signal lines or parameters. Thus, scheduling Common Utility Services cannot always be handled by Simulink alone. Each Common Utility Service has a setup block that allows the configuration of global parameters applicable to all feature block instances, and performs global actions to collect and reduce information provided by each feature instance block. Actions may include global consistency checks, memory allocation based on global feature usage, and build time calculation of global parameters that are based upon how individual feature blocks are used. The use of the global actions can greatly aid in the usability, run-time efficiency, and robustness of the service. In a purely hand-written code environment, often these global parameters must either be determined at run time (costing both RAM and ROM) or the user must manually track them.

Each Common Utility Service has a feature block that allows configuration of that particular feature instance. This service feature block is used to provide Simulink a location to provide or consume signals such that the rest of the Simulink model may be properly scheduled. The feature block generally provides the user with the main point for accessing the required functionality (e.g., receiving or transmitting a CAN message).

Some of the hand-written code behind the Common Utility Service blocks provides a single code call site for performing all or some of the processing across all feature blocks for that service. For example, a single C code function may be responsible for processing all the messages contained in a CAN port's receive buffers. However, an individual feature block may be receiving an individual CAN message; this allows distributed specification of the CAN messages to be received throughout a model. In this example, the single C code function would be called by the setup block, and the data made available to the rest of the Simulink model by individual CAN receive feature blocks.

In the above scenario, there is an implicit connection between the setup block and the individual feature blocks that is not represented by signal flow; thus, Simulink is not directly able to properly schedule the calling of the setup block relative to the individual feature blocks. To bridge this gap, an independent code buffering mechanism was added during code generation that allows scheduling code to execute before and/or after the normally generated Simulink code for each task. Additionally, the code buffering mechanism provides a priority parameter allowing ordering of code fragments submitted by different services. This mechanism is similar to the standard TLC Code Configuration functions that Real-Time Workshop provides (e.g., LibSetSourceFileCustomSection) [4]. The addition of priority allows simple scheduling of the submitted code fragments.

For the CAN example, the setup block can use the code buffer mechanism to ensure that all CAN packets have been processed and data made available to the CAN receive blocks prior to their execution in the normally generated model task code.

Figure 4 shows the data flow required to route data received in the CAN receive queue to the desired CAN block. An interrupt service routine receives and queues CAN messages. That receive queue is then periodically processed, routing data from the queue to each of the CAN receive blocks that require a particular message. Figure 5 shows a TLC excerpt used to create the task function. Figure 6 shows a TLC excerpt showing the registration of code used to periodically process the queue by the CAN setup block. Figure 7 shows the final generated code produced in the desired order.

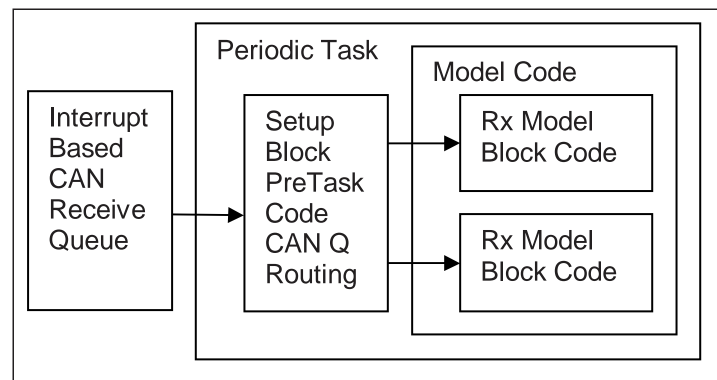

Figure 4. CAN receive data routing.

```
%%  Task infinite loop
while (TRUE)
{
  /* Wait until event occurs to start next
task iteration */
  rtos_event_pend(%<task.PendEvent>);

  %% Add service defined code buffers
  %%   Code to execute before Model's update
function
  %assign bufferName = "Task1_PreModelUpdate"
  %<LibCatCatBuffer_WriteBuffer(bufferName)>

  %% Execute Model Periodic Task
  %<FcnCallMdlStep(task.ModelTID)>

  %% Add service defined code buffers
  %%   Code to execute after Model's update
function
  %assign bufferName = "Task1_PostModelUpdate"
  %<LibCatCatBuffer_WriteBuffer(bufferName)>
}
```

Figure 5. Main task TLC.

```
%assign bufferName = "Task1_PreModelUpdate"
%assign priority = 0
%openfile codeBuffer
/* Setup Block Pre-Task Queue Routing Code */
...
...
%closefile codeBuffer
%<LibCatCatBuffer_AddToBuffer(bufferName,
priority, codeBuffer)>
```

Figure 6. CAN setup block pre model task code.

```
while (TRUE){
  /* Wait until event occurs to start next
task iteration */
  rtos_event_pend(%<task.PendEvent>);

  /* Setup Block Pre-Task Queue Routing Code
*/
  ...
  ...

  /* Execute Model's task function */
  model_task_step();
}
```

Figure 7. Generated task code.

A similar but often more complex ordering of loosely related services must occur at initialization. This ordering nearly always relies on information not contained in the Simulink signal flow, and thus cannot be scheduled directly by Simulink. Some services even require partial initialization prior to other services and then final initialization after. The same code buffer mechanism described above for use with periodic code may also be used to schedule Common Utility Service initializations.

An example of this could be the requirement to initialize the CAN service before the diagnostic service because the diagnostic service code may rely on the CAN service being initialize first in order for it to initialize its data link components.

Figure 8 shows TLC code used to create the main initialization function. Figure 9 shows TLC code used to register the diagnostic service initialization code. Figure 10 shows the TLC code used to register the CAN service initialization code. Finally, Figure 11 shows the generated initialization code generated in the desired order.

```
static void startup_init(void)
{
  %% Add service defined buffers
  %%   Initialization code to execute
  %%   BEFORE the model initialization has
occurred

%<LibCatCatBuffer_WriteBuffer("InitPreModel")>

  /* Call Model's Initialization Function */
  %<LibCallModelInitialize()>

  %% Add service defined buffers
  %%   Initialization code to execute
  %%   AFTER the model intialization has
occurred

%<LibCatCatBuffer_WriteBuffer("InitPostModel")
>
}
```

Figure 8. Initialization function TLC.

```
%assign bufferName = "InitPreModel"
%assign priority = 1
%openfile codeBuffer
/* Diagnostics Initialization Code */
...
...
%closefile codeBuffer
%<LibCatCatBuffer_AddToBuffer(bufferName,
priority, codeBuffer)>
```

Figure 9. Diagnostic initialization registration.

```
%assign bufferName = "InitPreModel"
%assign priority = 0
%openfile codeBuffer
/* CAN Initialization Code */
...
...
%closefile codeBuffer
%<LibCatCatBuffer_AddToBuffer(bufferName,
priority, codeBuffer)>
```

Figure 10. CAN initialization registration.

```
static void startup_init(void)
{
  /* CAN Initialization Code */
  ...
  ...

  /* Diagnostics Initialization Code */
  ...
  ...

  /* Call Model's Initialization Function */
  model_initialize();
}
```

Figure 11. Generated initialization code.

INTEGRATION WITH MODEL REFERENCE - Model Reference provides the ability to break down a model into subcomponents to improve performance and also to facilitate configuration management. Each component's code can be incrementally generated and compiled. Using Model Reference can reduce the time needed for simulation and code generation for large Simulink applications.

For implementation of Common Utility Services, Model Reference introduces some barriers and challenges. First, getting a complete view of information across the entire Simulink application can be difficult. Assuming that the top-level model centrally configures the Common Utility Services, child reference models can pass data from the individual service instances up to the top model using model reference user data [5]. A second challenge is that using model reference with Common Utility Services could add significant run-time overhead if the Common Utility Services are implemented in such a way that a large number of parameters must be passed across the stack from the referencing model to the referenced model.

CONFIGURATION OF COMMON UTILITY SERVICES - There are several methods for configuring settings and data for individual interface block instances in the Simulink model. Settings can be configured directly in the individual block masks, through variables and parameters stored in the MATLAB workspace, or they can be stored in an external tool. Each method has benefits and drawbacks.

Block-Based Configuration - Configuring settings directly in the block is the most common method for configuring block parameters in Simulink. Entering values directly into block masks can make a diagram easy to read, and easy to modify. When creating or studying the model's algorithms, parameters can be changed directly in the diagram with very few steps.

There are two clear downsides to block-based configuration. One is that the data is stored within the model, meaning that the model file itself must be modified if values change. Additionally, entering parameter values into blocks directly prevents the user from easily swapping out parameter sets, a task that is much easier with workspace-based configuration. A second downside is that the parameters cannot be viewed easily in a single place. Placing the parameters in an M-file or in the MATLAB workspace allows the user to review all the data in a flat-structured, single location.

Workspace-Based Configuration - Referencing block parameters from the MATLAB workspace offers the ability to decouple a Simulink application from its configuration data. A key benefit of this method is the ability to have multiple sets of swappable configuration data for the same application model. There are two methods that that will be examined to parameterize blocks in the workspace: multiple individual configuration references and Simulink.Parameter derived objects that contain multiple configuration options as a single variable reference.

The simplest method is to allow each individual Common Utility Service parameter to be referenced from the MATLAB workspace. This method is trivial to set up, and the parameters can be stored in an M-file or MAT-file that gets loaded into the workspace. Most standard Simulink blocks use this method of configuration. This method allows each individual configuration to be chosen in a standard block mask. Each individual configuration may be entered as literal value, a standard variable reference, or as a Simulink.Parameter reference. While this method allows easy sharing of references to individual common configuration parameters, a downside is that with many instances of a service block, it can become difficult to manage the sheer number of parameters this creates. Additionally, if the mask is configured with combo boxes or check boxes to make working with the mask easier, there is no support for referencing workspace parameters.

An example of the usage of individual parameters to configure a service is illustrated in Figure 12 below. Individual configuration parameters can be entered directly in the block, via standard workspace parameters or via Simulink.Parameter object references.

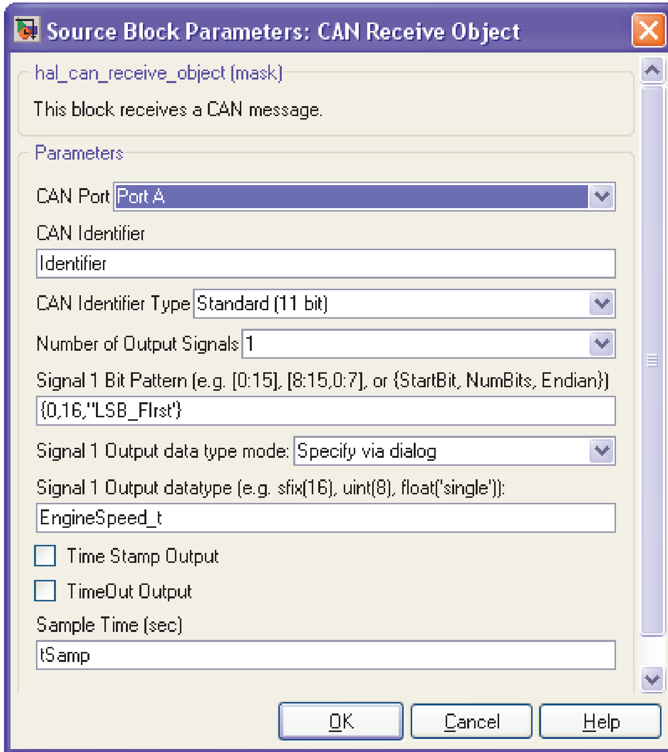

Figure 12. CAN Receive block using block and workspace-based configuration.

A more complex but comprehensive method is to configure blocks via a single (or relatively few) workspace objects derived from Simulink.Parameter. In this case, each Simulink.Parameter derived object will contain multiple configuration parameters used by the service. Typically, Simulink.Parameter objects are created with object fields that describe a single configuration parameter. This method uses the fields of a single Simulink.Parameter object to describe multiple configuration parameters of the service. By creating custom objects that contain the properties necessary to parameterize a service block, the properties and dialog for parameterizing a given block are encapsulated into a single workspace object that is defined separately from the model. The block mask no longer contains a GUI to adjust each of the service's configuration parameters. A GUI is provided for the object through the Simulink Model Explorer and customized object GUIs.

One downside of using objects to parameterize the interface blocks is that the objects require significantly more effort to create and maintain. Using objects is also more complex than just typing parameter values directly into a block's mask, which may be a deterrent to new users.

An example of the usage of a parameter object to specify multiple service configuration parameters can be seen for another version of a CAN receive block below. Figure 13 shows the CAN receive block's mask that allows configuring the block. This particular block contains two object references as well as a standard parameter that is entered directly in the mask. Figure 14, shows instantiation of the object in the MATLAB workspace. Finally, Figure 15 shows the GUI used to edit the object.

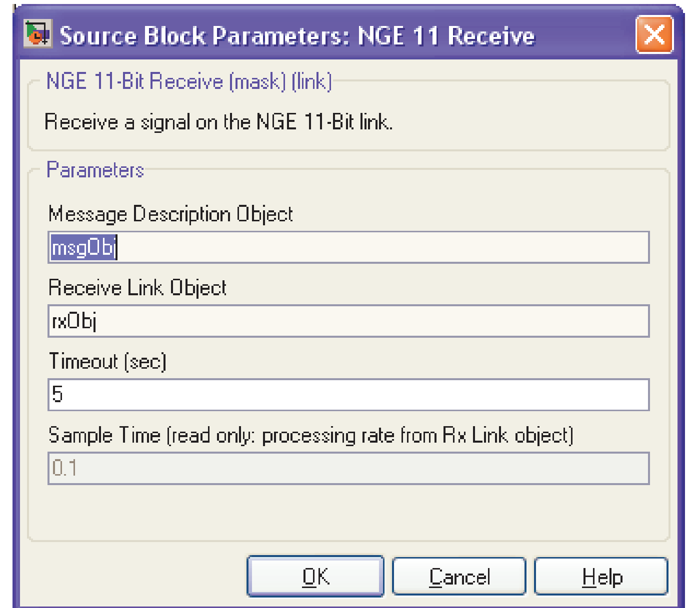

Figure 13. CAN Receive block mask using parameter object references containing multiple service configuration parameters.

```
>> msgObj = Cat.MsgDescription

Cat.MsgDescription
           MessageIdentifier:   'Message Name'
                  NumSignals:   1
                 SignalArray:   [1x1 Cat.MsgSignal]
```

Figure 14. MATLAB object instantiation.

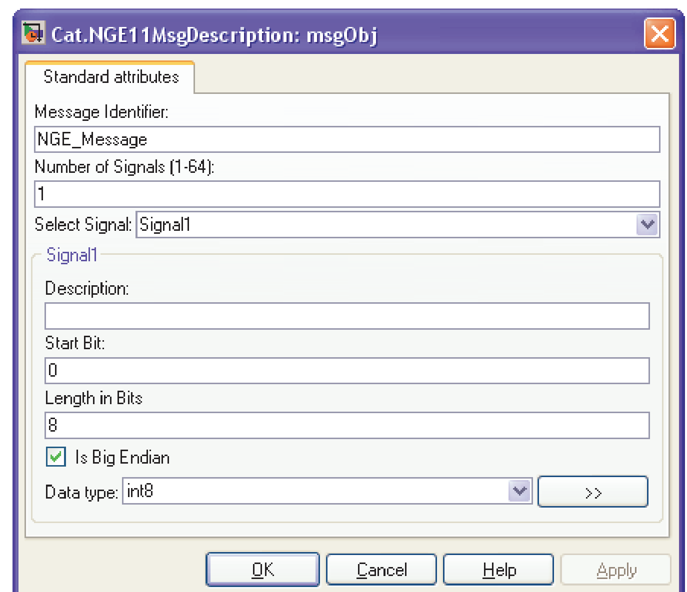

Figure 15. Parameter object GUI.

<u>External-Based Configuration</u> - In some cases, engineers who configure Common Utility Services may want to set up the services' configuration, but not want to interact directly with the model. A tool outside of

Simulink may be desirable in this case. An outside tool may provide additional functionality, and may also be accessible to more users. However, an outside tool will require writing an interface to MATLAB and Simulink, and may not have the seamless level of integration that can be obtained using block-based or workspace-based configuration.

Configuration Recommendations - Many engineers, particularly those who are new to Simulink, will start by configuring parameters in blocks directly. It has also been commonly observed that research engineers tend to configure data directly in blocks, while production engineers usually configure block data through workspace references. If the Common Utility Services are to be broadly used by research and production engineers, ideally both block-based and workspace-based configuration methods could be used.

Currently, all methods described have been used for various Common Utility Service blocks: block-based configuration, individual workspace variable configuration, and Simulink.Parameter object configuration. Finding a solution that supports all configuration methods without excessive maintenance overhead continues to be an area of development.

CONCLUSION

With increased use of code generation with Simulink while integrating Common Utility Services, services are easier to use and more efficient to implement. Implementing Common Utility Services in Simulink facilitates unifying the development process from simulation, rapid prototyping, and production, providing a common method to access Common Utility Services throughout the process while improving both development and execution efficiency.

REFERENCES

1. Thate, Jeffrey M., Kendrick, Larry E., Nadarajah, Siva, Caterpillar Automatic Code Generation, presented at SAE World Congress, 2004-01-0894, 2004.

2. xPC Target Documentation, “CAN I/O Support,” The MathWorks, Inc.

3. xPC Target product page, The MathWorks, Inc. http://www.mathworks.com/products/xpctarget/

4. Real-Time Workshop Documentation, “Target Language Compiler (TLC),” The MathWorks, Inc.

5. Real-Time Workshop Documentation , “Advanced Functions, TLC Function Library Reference, LibGetModelReferenceUserData(),” The MathWorks, Inc.

CONTACT

Jeffrey M. Thate
Caterpillar, Inc.
PO Box 610
Mossville, IL 61552-0610
E-mail: thatejm@cat.com

Robert A. Kagy
Caterpillar, Inc.
PO Box 610
Mossville, IL 61552-0610
E-mail: KAGY_ROBERT_A@cat.com

Robyn A. Jackey
Senior Technical Consultant
The MathWorks
39555 Orchard Hill Place
Suite 280
Novi, MI 48375
E-mail: robyn.jackey@mathworks.com

Roger Theyyunni
Senior Technical Consultant
The MathWorks
39555 Orchard Hill Place
Suite 280
Novi, MI 48375
E-mail: roger.theyyunni@mathworks.com

Jagadish Gattu
Technical Consultant
The MathWorks
3 Apple Hill Drive
Natick, MA 01760
E-mail: jagadish.gattu@mathworks.com

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.