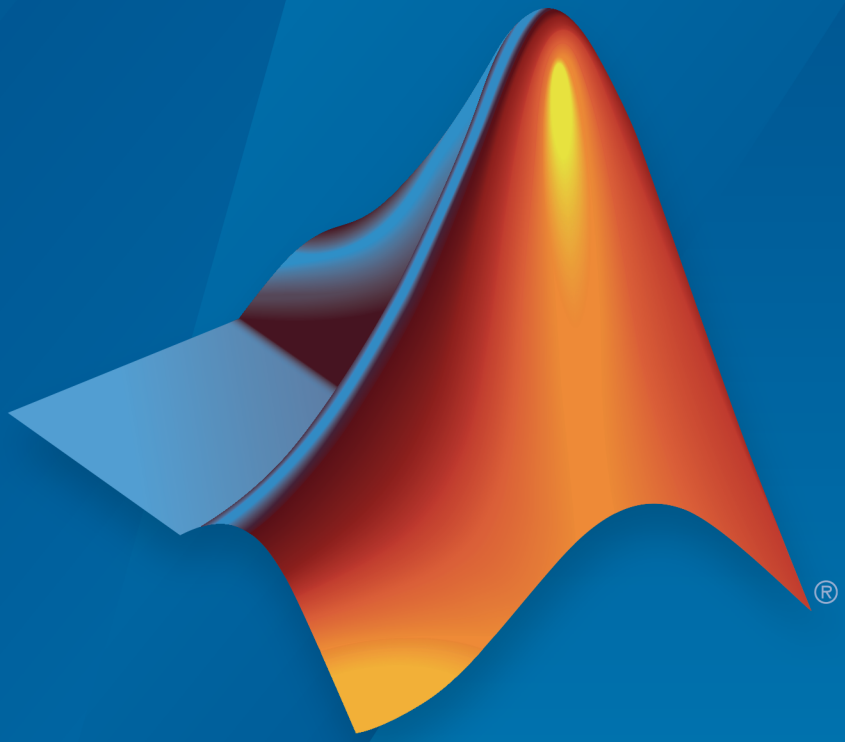
Simulink® Coder™

Target Language Compiler



MATLAB® & SIMULINK®

R2015a

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

Simulink® Coder™ Target Language Compiler

Revision History

April 2011	Online only	New for Version 8.0 (Release 2011a)
September 2011	Online only	Revised for Version 8.1 (Release 2011b)
March 2012	Online only	Revised for Version 8.2 (Release 2012a)
September 2012	Online only	Revised for Version 8.3 (Release 2012b)
March 2013	Online only	Revised for Version 8.4 (Release 2013a)
September 2013	Online only	Revised for Version 8.5 (Release 2013b)
March 2014	Online only	Revised for Version 8.6 (Release 2014a)
October 2014	Online only	Revised for Version 8.7 (Release 2014b)
March 2015	Online only	Revised for Version 8.8 (Release 2015a)

Check Bug Reports for Issues and Fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

Contents

1 What is the Target Language Compiler?

2 Getting Started

3 Target Language Compiler Tutorials

4 Code Generation Architecture

5 *model*.rtw File and Authoring S-Functions and Data Objects

6 Directives and Built-In Functions

7 Debugging TLC Files

8 Inlining S-Functions

9 TLC Function Library Reference

A TLC Error Handling

1

What is the Target Language Compiler?

Introduction to the Target Language Compiler

In this section...

Target Language Compiler Overview

Target Language Compiler (TLC) is an integral part of the Simulink® Coder™ product. It enables you to customize generated code. Through customization, you can produce platform-specific code, or you can incorporate your own algorithmic changes for performance, code size, or compatibility with existing methods that you prefer to maintain.

The TLC includes:

- A set of TLC files corresponding to a subset of the provided Simulink blocks.
- TLC files for model-wide information that specify header and parameter information.

The TLC files are ASCII files that explicitly control the way code is generated. By editing a TLC file, you can alter the way code is generated.

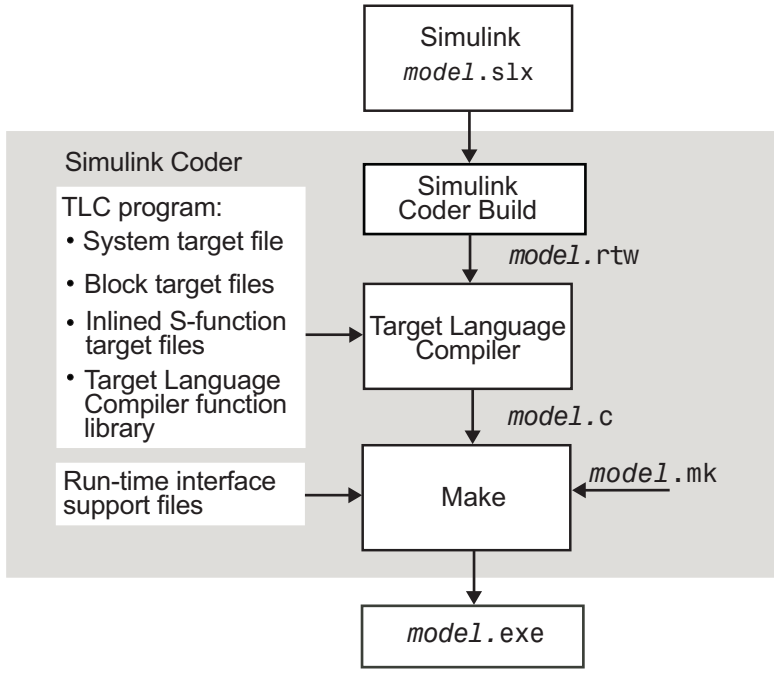
The Target Language Compiler provides a complete set of ready-to-use TLC files for generating ANSI® C or C++ code. You can view the TLC files and make minor — or extensive — changes to them. This open environment gives you tremendous flexibility when it comes to customizing the generated code.

For more information, see "C/C++ S-Functions" which explains how to write wrapped and fully inlined S-functions, with a special emphasis on the `mdlRTW()` function.

Note: You should not customize TLC files in the folder *`matlabroot`*`/rtw/c/tlc` even though the capability exists to do so. Such TLC customizations might not be applied during the code generation process and can lead to unpredictable results.

Overview of the TLC Process

This top-level diagram shows how the Target Language Compiler fits in with the code generation process.



The Target Language Compiler (TLC) is designed for one purpose — to convert the model description file `model.rtw` (or similar files) into target-specific code or text.

The Target Language Compiler transforms an intermediate form of a Simulink block diagram, called `model.rtw`, into C or C++ code. The `model.rtw` file contains a "compiled" representation of the model describing the execution semantics of the block diagram in a very high-level language. For more information, see "Introduction to the model.rtw File".

The word *target* in Target Language Compiler refers not only to the high-level language to be output, but also to the nature of the real-time system on which the code will be executed. TLC-generated code is thus able to respect and exploit the capabilities and limitations of specific processor architectures (the target).

After reading the *model*.rtw file, the Target Language Compiler generates its code based on *target files*, which specify particular code for each block, and *model-wide files*, which specify the overall code style. The TLC works like a text processor, using the target files and the *model*.rtw file to generate ANSI C or C++ code.

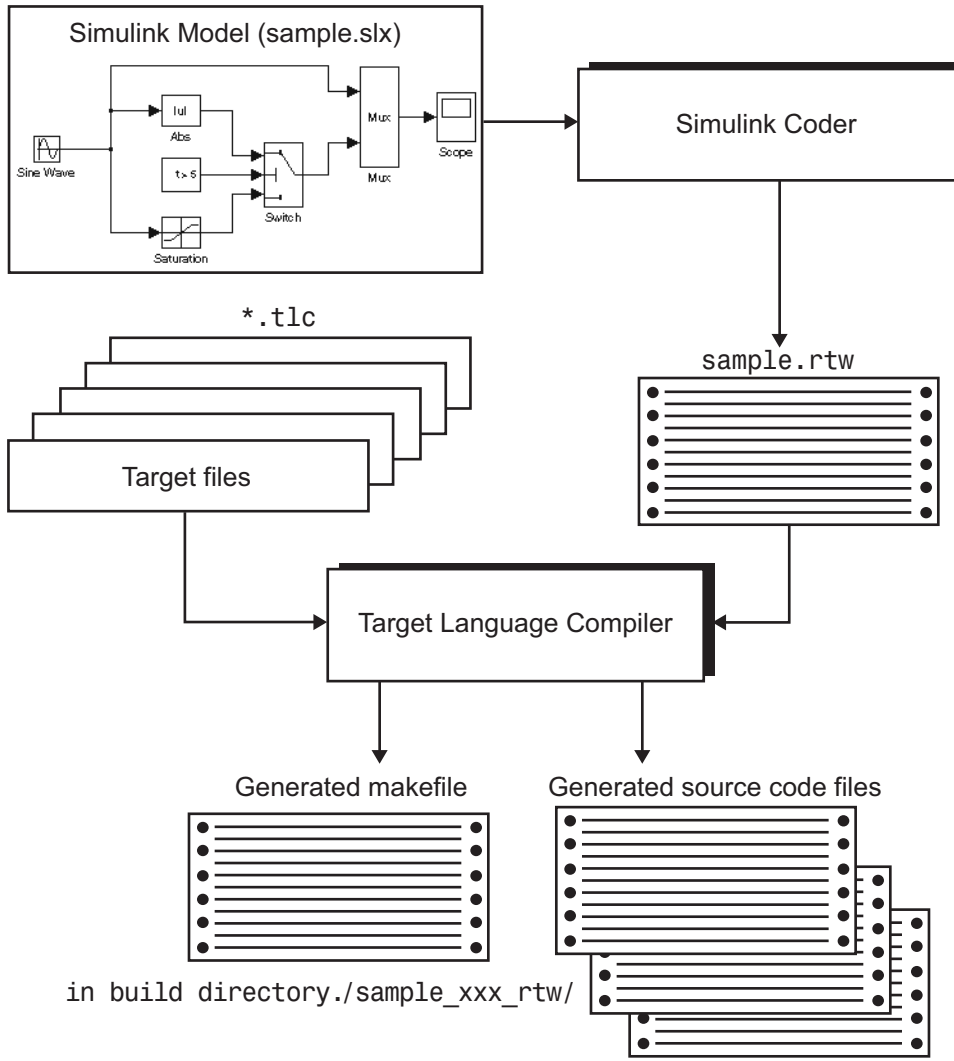
To create a target-specific application, the code generator requires a template makefile that specifies a C or C++ compiler and compiler options for the build process. The code generator transforms the template makefile into a target makefile (*model*.mk) by performing token expansion specific to a given model. The target makefile is a modified version of the generic rt_main file (or grt_main), which you must modify to conform to the target's specific requirements, such as interrupt service routines. See "Template Makefiles and Make Options" and "Customize Template Makefiles" for more information about template makefiles.

The Target Language Compiler has similarities with HTML, Perl, and MATLAB®. It has markup syntax similar to HTML, the power and flexibility of Perl and other scripting languages, and the data handling power of MATLAB (TLC can invoke MATLAB functions). The code generated by TLC is highly optimized and fully commented, and can be generated from linear, nonlinear, continuous, discrete, or hybrid Simulink models. The models can include Simulink blocks that are automatically converted to code, with the exception of MATLAB function blocks and S-function blocks that invoke MATLAB files. The Target Language Compiler uses *block target files* to transform each block in the *model*.rtw file and a *model-wide target file* for global customization of the code.

You can incorporate C MEX S-functions, along with the generated code, into the program executable. You can also write a target file for your C MEX S-function to *inline* the S-function (see "Inline C MEX S-Functions" on page 8-5), thus potentially improving performance by eliminating function calls to the S-function itself and the memory overhead of theSimStruct of the S-function. Inlining an S-function incorporates the S-function block's code into the generated code for the model. When a TLC target file is not present for the S-function, its C or C++ code file is invoked via a function call. For more information on inlining S-functions, see "S-Function Inlining". You can also write target files for MATLAB language files or Fortran S-functions.

Overview of the Code Generation Process

The following figure shows how the Target Language Compiler works with its target files and the Simulink Coder code generator output to produce code.



When generating code from a Simulink model, the first step in the automated process is to generate a *`model`*`.rtw` file. The *`model`*`.rtw` file includes the model-specific information required for generating code from the Simulink model. *`model`*`.rtw` is passed to the Target Language Compiler, which uses it in combination with a set of included system target files and block target files to generate the code.

Only the final executable file is written directly to the current folder. For other files created during code generation, including the *model*.rtw file, a build folder is used. This folder is created in the current folder and is named *model*_*target*_rtw, where *target* is the abbreviation for the target environment, e.g., grt for the generic real-time target.

Files placed in the build folder include

- The body for the generated C or C++ source code (*model*.c or *model*.cpp)
- Header files (*model*.h)
- Header file *model*_private.h defining parameters and data structures private to the generated code
- A makefile, *model*.mk, for building the application
- Additional files, described in "Files and Folders Created by Build Process"

Target Language Compiler Capabilities

In this section...

Why Use TLC?

If you simply need to produce ANSI C or C++ code from Simulink models, you do not need to know how to prepare files for the Target Language Compiler. If you need to customize the output, you must run the Target Language Compiler. Use the Target Language Compiler if you need to

- Customize the set of options specified by your system target file
- Inline the code for S-Function blocks
- Generate additional or different types of files

Both the MATLAB Function block and the Embedded Coder® product facilitate code customization in a variety of ways. You might be able to accomplish what you need with them, without the need to write TLC files. However, you do need to prepare TLC files if you intend to inline S-functions.

Customizing Output

To produce customized output using the Target Language Compiler, it helps if you understand how blocks perform their functions, what data types are being manipulated, the structure of the `model.rtw` file, and how to modify target files to produce the desired output. "Directives and Built-In Functions" describes the target language directives and their associated constructs. You will use the Target Language Compiler directives and constructs to modify existing target files or create new ones, depending on your needs. See "TLC Files" on page 4-15 for more information about target files.

Note: You should not customize TLC files in the folder `matlabroot/rtw/c/tlc` even though the capability exists to do so. Such TLC customizations might not be applied during the code generation process and can lead to unpredictable results.

Inlining S-Functions

The Target Language Compiler provides a great deal of freedom for altering, optimizing, and enhancing the generated code. One of the most important TLC features is that it lets you inline S-functions that you write to add your own algorithms, device drivers, and custom blocks to a Simulink model.

To create an S-function, you write code following a well-defined application program interface (API). By default, the Target Language Compiler will generate noninlined code for S-functions that invokes them using this same API. This generalized interface incurs a fair amount of overhead due to the presence of a large data structure called the `SimStruct` for each instance of each S-Function block in your model. In addition, extra run-time overhead is involved whenever methods (functions) within your S-function are called. You can eliminate this overhead by using the Target Language Compiler to inline the S-function, by creating a TLC file named *`sfunction_name`*`.tlc` that generates source code for the S-function as if it were a built-in block. Inlining an S-function improves the efficiency of the generated code and reduces memory usage.

Modifying and Diversifying Code Generation

In principle, you can use the Target Language Compiler to convert the *`model`*`.rtw` file into another form of output (for example, OODBMS objects) by replacing the supplied TLC files for each block it uses. Likewise, you can also replace some or all of the shipping system-wide TLC files, though doing so is not a recommended practice. To maintain such customizations, you might need to update your TLC files with each Simulink Coder product release. MathWorks continues to improve code generation by adding features, improving efficiency, and altering the contents of *`model`*`.rtw`. Inlined TLC files that you create, on the other hand, generally are backward compatible, provided that they invoke only documented TLC library and built-in functions.

Code Generation Process

In this section...

Process Overview

The code generator invokes the Target Language Compiler after a model is compiled into an intermediate form (*`model`*`.rtw`) suitable for code generation. To generate code, the Target Language Compiler uses its library of functions to transform two classes of target files:

- System target files
- Block target files

System target files are used to specify the overall structure of the generated code, tailoring for specific target environments. Block target files are used to implement the functionality of Simulink blocks, including user-defined S-function blocks.

You can create block target files for C MEX, Fortran, and MATLAB language S-functions to fully inline block functionality into the body of the generated code. C MEX S-functions can be noninlined, wrapper-inlined, or fully inlined. Fortran S-functions must be wrapper-inlined or fully inlined.

How TLC Determines S-Function Inlining Status

Whenever the Target Language Compiler encounters an entry for an S-function block in the *`model`*`.rtw` file, it must decide whether to generate a call to the S-function or to inline it.

Because they cannot use `SimStructs`, Fortran and MATLAB language S-functions must be inlined. This inlining can either be in the form of a full block target file or a one-line block target file that refers to a substitute C MEX S-function source file.

The Target Language Compiler selects a C MEX S-function for inlining if an explicit `mdlRTW()` function exists in the S-function code or if a target file for the current target

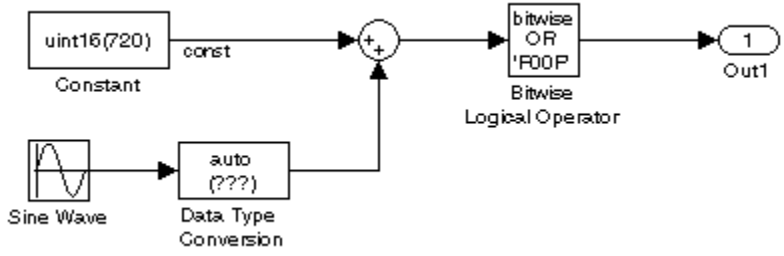
language for the current block is in the TLC file search path. If a C MEX S-function has an explicit `mdlRTW()` function, there must be a corresponding target file or an error condition results.

The target file for an S-function must have the same root name as the S-function and must have the extension `.tlc`. For example, the C MEX S-function `sfix_bitop` uses these files, which are available in *`matlabroot`*`/toolbox/simulink/fixedandfloat/`:

Location and Filename(s)	Purpose
`sfix_bitop.c`	C source file
`sfix_bitop.mex*`	Compiled files
`tlc_c/sfix_bitop.tlc`	TLC target file

A Look at Inlined and Noninlined S-Function Code

This example focuses on the C MEX S-function `sfix_bitop` in *`matlabroot`*`/toolbox/simulink/fixedandfloat/sfix_bitop.c`. The code generation options are set to allow reuse of signal memory for signal lines that were not set as tunable signals.



The code generated for the bit-wise operator block reuses a temporary variable that is set up for the output of the sum block to save memory. This results in one very efficient line of code, as seen here.

```
/* Bitwise Logic Block: <Root>/Bitwise Logical Operator */
/* [input] OR 'F00F' */
rtb_temp2 |= 0xF00F;
```

Initialization or setup code is not required for this inlined block.

If this block were not inlined, the source code for the S-function itself with its various options would be added to the generated code base, memory would be allocated in the generated code for the block's `SimStruct` data, and calls to the S-function methods would be generated to initialize, run, and terminate the S-function code. To execute the `mdlOutputs` function of the S-function, code would be generated like this:

```
/* Level2 S-Function Block: <Root>/Bitwise Logical Operator (sfix_bitop) */
  {
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnOutputs(rts, tid);
  }
```

The entire `mdlOutputs` function is called and runs just as it does during simulation. That's not everything, though. There is also registration, initialization, and termination code for the noninlined S-function. The initialization and termination calls are similar to the fragment above. Then, the registration code for an S-function with just one inport and one outport is 72 lines of C code generated as part of file *`model`*`_reg.h`.

```
/*Level2 S-Function Block: <Root>/Bitwise Logical Operator (sfix_bitop) */
    {
      extern void untitled_sf(SimStruct *rts);
      SimStruct *rts = ssGetSFunction(rtS, 0);

      /* timing info */
      static time_T sfcnPeriod[1];
      static time_T sfcnOffset[1];
      static int_T sfcnTsMap[1];

      {
        int_T i;

        for(i = 0; i < 1; i++) {
          sfcnPeriod[i] = sfcnOffset[i] = 0.0;
        }
      }
      ssSetSampleTimePtr(rts, &sfcnPeriod[0]);
      ssSetOffsetTimePtr(rts, &sfcnOffset[0]);
      ssSetSampleTimeTaskIDPtr(rts, sfcnTsMap);
      ssSetMdlInfoPtr(rts, ssGetMdlInfoPtr(rtS));

      /* inputs */
      {
        static struct _ssPortInputs inputPortInfo[1];

        _ssSetNumInputPorts(rts, 1);
        ssSetPortInfoForInputs(rts, &inputPortInfo[0]);

        /* port 0 */
        {

          static real_T const *sfcnUPtrs[1];
          sfcnUPtrs[0] = &rtU.In1;
```

```
            ssSetInputPortSignalPtrs(rts, 0, (InputPtrsType)&sfcnUPtrs[0]);
            _ssSetInputPortNumDimensions(rts, 0, 1);
            ssSetInputPortWidth(rts, 0, 1);
          }
        }
.
.
.
```

This continues until S-function sizes and methods are declared, allocated, and initialized. The amount of registration code generated is essentially proportional to the number and size of the input ports and output ports.

A noninlined S-function will typically have a significant impact on the size of the generated code, whereas an inlined S-function can be close to the handwritten size and performance of the generated code.

The Advantages of Inlining S-Functions

In this section...

Goals

The goals of generated code usually include compactness and speed. On the other hand, S-functions are run-time-loadable extension modules for adding block-level functionality to Simulink. As such, the S-function interface is optimized for flexibility in configuring and using blocks in a simulation environment with capability to allow run-time changes to a block's operation via parameters. These changes typically take the form of algorithm selection and numerical constants for the block algorithms.

While switching algorithms is a desirable feature in the design phase of a system, when the time comes to generate code, this type of flexibility is often dropped in favor of optimal calculation speed and code size. The Target Language Compiler was designed to allow the generation of code that is compact and fast by selectively generating only the code you need for one instance of a block's parameter set.

When To Avoid Inlining

You might decide not to inline C MEX S-functions that have

- Few or no numerical parameters
- One algorithm that is already fixed in capability. For example, it has no optional modes or alternate algorithms.
- Support for only one data type
- A significant or large code size in the `mdlOutputs()` function
- Multiple instances of this block in your models

Whenever you encounter this situation, the effort of inlining the block might not improve execution speed and could actually increase the size of the generated code. The tradeoff is

in the size of the block's body code generated for each instance versus the size of the child SimStruct created for each instance of a noninlined S-function in the generated code.

Alternatively, you can use a hybrid inlining method known as a C MEX wrapped S-function, where the block target file simply generates a call to a custom code function that the S-function itself also calls. This approach might be the optimal solution for code generation in the case of a large piece of existing code. See "S-Function Inlining" for the procedure and an example of a wrapped S-function.

Inlining Process

The strategy for improving code from blocks centers on determining what part of a block's operations are active and used in the generated code and what parts can be predetermined or left out.

In practice, this means the TLC code in the block target file will select an algorithm that is a subset of the algorithms contained in the S-function itself and then selectively hard-code numerical parameters that are not to be changed at run time. This reduces code memory size and results in code that is often much faster than its S-function counterpart when mode selection is a significant part of S-function processing. Additionally, function-call overhead is eliminated for inlined S-functions, as the code is generated directly in the body of the code unless there is an explicit call to a library function in the generated code.

The algorithm selections and parameters for each block are output in the initial phase of the code generation process from the registered S-function parameter set or the `mdlRTW()` function (if present), which results in entries in the model's `.rtw` file for that block at code generation time. A file written in the target language for the block is then called to read the entries in the *`model`*`.rtw` file and compute the generated code for this instance of the block. This TLC code is contained in the block target file.

One special case for inlined S-functions is for the case of I/O blocks and drivers such as A/D converters or communications ports. For simulation, the I/O driver is typically coded in the S-function as a pure source, a pass-through, or a pure sink. In the generated code, however, an actual interface to the I/O device must be made, typically through direct coding with the common `_in()`, `_out()` functions, inlined assembly code, or a specific set of I/O library calls unique to the device and target environment.

Search Algorithm for Locating TLC Files

The Target Language Compiler uses the following search order to locate TLC files:

1. Current folder.
2. Locations specified by `%addincludepath` directives. The compiler evaluates multiple `%addincludepath` directives from the *bottom up*.
3. Locations specified by `-I` options. The compiler evaluates multiple `-I` options from *right to left*.

For inlined S-functions TLC files, the Simulink Coder build process supports the following locations:

- The folder where the S-function executable (MEX or `.m`) file is located.
- S-function folder subfolder `./tlc_c` (for C or C++ language targets).
- The current folder when the Simulink Coder build process is initiated.

Note: Note: Placing the inlined S-function TLC file elsewhere is not supported, even if the location is in the TLC include path.

The first target file encountered with the required name that implements the specified language is used in processing the S-function *`model`*`.rtw` file entry.

Note: The compiler does *not* search the MATLAB path, and will not find a file that is available only on that path. The compiler searches only the locations described above.

Availability for Inlining and Noninlining

S-functions can be written in MATLAB language, Fortran, C, and C++. TLC inlining of S-functions is available as indicated in this table.

Inline TLC Support by S-Function Type

S-Function Type	Noninlining Supported	Inlining Supported
MATLAB language	No	Yes
Fortran MEX	No	Yes
C	Yes	Yes
C++	Yes	Yes

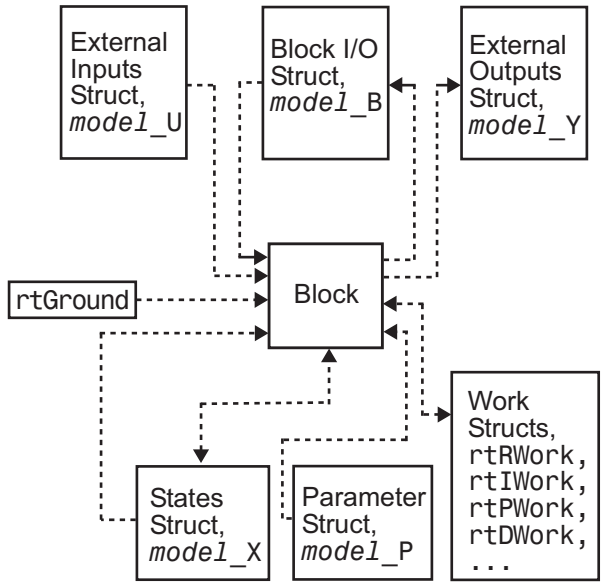
Getting Started

Code Architecture

Before investigating the specific code generation pieces of the Target Language Compiler (TLC), consider how Target Language Compiler generates code for a simple model. From the next figure, you see that blocks place code into `Mdl` routines. This shows `MdlOutputs`.


```
static void simple_output(int_T tid)
{

  /* Sin Block: '<Root>/Sine Wave' */

  simple_B.SineWave_d = simple_P.SineWave_Amp *
    sin(simple_P.SineWave_Freq * simple_M->Timing.t[0] +
    simple_P.SineWave_Phase) + simple_P.SineWave_Bias;

  /* Gain: '<Root>/Gain' */
  simple_B.Gain_d = simple_B.SineWave_d * simple_P.Gain_Gain;

  /* Outport: '<Root>/Out1' */
  simple_Y.Out1 = simple_B.Gain_d;
}
```

Blocks have inputs, outputs, parameters, states, plus other general properties. For example, block inputs and outputs are generally written to a block I/O structure (generated with identifiers of the type *`model`*`_B`), where *`model`* is the model name). Block inputs can also come from the external input structure (*`model`*`_U`) or the state structure when connected to a state port of an integrator (*`model`*`_X`), or ground (`rtGround`) if unconnected or grounded. Block outputs can also go to the external output structure, (*`model`*`_Y`). The following diagram shows the general block data mappings.



This discussion should give you a general sense of what the block object looks like. Now, you can look at specific pieces of the code generation process that are specific to the Target Language Compiler.

Target Language Compiler Overview

In this section...

The Target Language Compiler Process

To write TLC code for your S-function, you need to understand the Target Language Compiler process for code generation. As previously described, the Simulink software generates a *model*`.rtw` file that contains a high-level representation of the execution semantics of the block diagram. The *model*`.rtw` file is an ASCII file that contains a data structure in the form of a nested set of TLC records. The records comprise property name/property value pairs. The Target Language Compiler reads the *model*`.rtw` file and converts it into an internal representation.

Next, the Target Language Compiler runs (interprets) the TLC files, starting first with the system target file, for example, `grt.tlc`. This is the entry point to the system TLC and block files, that is, other TLC files included in or generated from the TLC file passed to Target Language Compiler on its command line (`grt.tlc`). As the TLC code in the system and block target files is run, it uses, appends to, and modifies the existing property name/property value pairs and records initially loaded from the *model*`.rtw` file.

model.rtw Structure

The structure of the *model*`.rtw` file mirrors the block diagram's structure:

- For each nonvirtual system in the model, there is a corresponding system record in the *model*`.rtw` file.
- For each nonvirtual block within a nonvirtual system, there is a block record in the *model*`.rtw` file in the corresponding system.

The basic structure of *model*`.rtw` is

```
CompiledModel {
  System {
    Block {
      DataInputPort {
        ...
```

```
      }
      DataOutputPort{
        ...
      }
      ParamSettings {
        ...
      }
      Parameter {
        ...
      }
    }
  }
}
```

Operating Sequence

For each occurrence of a given block in the model, a corresponding block record exists in the *model*`.rtw` file. The system target file TLC code loops through block records and calls the functions in the corresponding block target file for that block type. For inlined S-functions, it calls the inlining TLC file.

There is a method for getting block-specific information (internal block information, as opposed to inputs, outputs, parameters, etc.) into the block record in the *model*`.rtw` file for a block by using the `mdlRTW` function in the C MEX function of the block.

Among other things, the `mdlRTW` function allows you to write out parameter settings (`ParamSettings`), that is, unique information pertaining to this block. For parameter settings in the block TLC file, direct accesses to these fields are made from the block TLC code and can be used to alter the generated code as desired.

Inlining S-Functions

In this section...

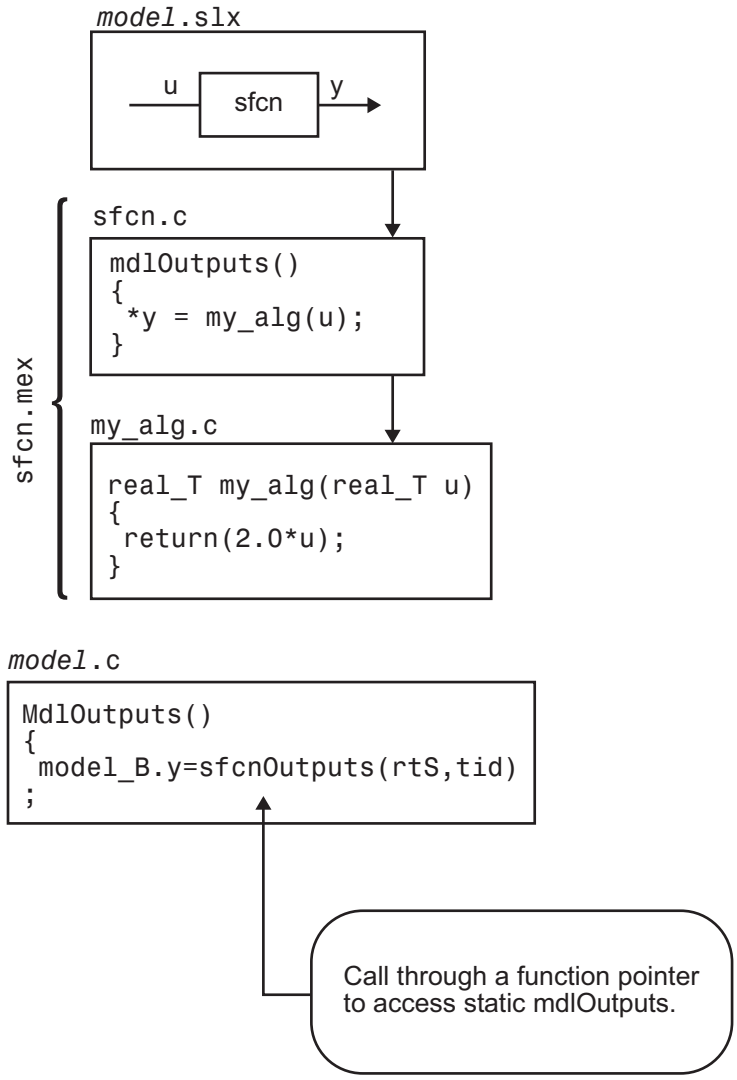
Inlining an S-function

To inline an S-function means to provide a TLC file for an S-Function block that will replace the C, C++, Fortran, or MATLAB language version of the block that was used during simulation.

Noninlined S-Function

If an inlining TLC file is not provided, most targets support the block by recompiling the C MEX S-function for the block. As discussed earlier, there is overhead in memory usage and speed when using a C/C++ coded S-function and a limited subset of `mx*` API calls supported within the code generator context. If you want the most efficient generated code, you must inline S-functions by writing a TLC file for them.

When the simulation needs to execute one of the functions for an S-function block, it calls the MEX-file for that function. When the code generator executes a noninlined S-function, it does so in a similar manner, as this diagram illustrates.



Types of Inlining

It is helpful to define two categories of inlining:

- Fully inlined S-functions

- Wrapper inlined S-functions

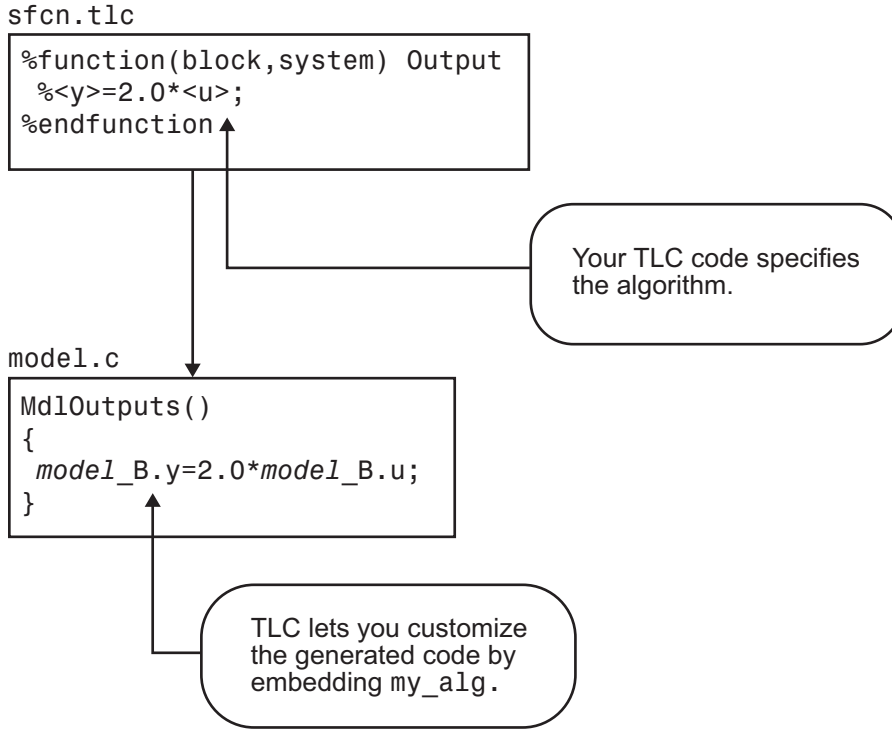
While both inline the S-function and remove the overhead of a noninlined S-function, the two approaches are different. The first example below, using `timestwo.tlc`, is considered a fully inlined TLC file, where the full implementation of the block is contained in the TLC file for the block.

The second example uses a wrapper TLC file. Instead of generating the algorithmic code in place, this example calls a C function that contains the body of code. There are several potential benefits for using the wrapper TLC file:

- It provides a way for the C MEX S-function and the generated code to share the C code. You do not need to write the code twice.
- The called C function is an optimized routine.
- Several of the blocks might exist in the model, and it is more efficient in terms of code size to have them call a function, as opposed to each creating identical algorithmic code.
- It provides a way to incorporate legacy C code seamlessly into generated code.

Fully Inlined S-Function Example

Inlining an S-function provides a mechanism to directly embed code for an S-function block into the generated code for a model. Instead of calling into a separate source file via function pointers and maintaining a separate data structure (`SimStruct`) for it, the code appears “inlined” as the next figure shows.



The S-function `timestwo.c` provides a simple example of a fully inlined S-function. This block multiplies its input by 2 and outputs it. The C MEX version of the block is in *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/src/timestwo.c`, and the inlining TLC file for the block is in *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/tlc_c/timestwo.tlc`.

timestwo.tlc

```
%implements "timestwo" "C"

%% Function: Outputs ========================================
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %%
  /* Multiply input by two */
  %assign rollVars = ["U", "Y"]
  %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
    %<LibBlockOutputSignal(0, "", lcv, idx)> = \
    %<LibBlockInputSignal(0, "", lcv, idx)> * 2.0;
```

```
  %endroll
%endfunction
```

TLC Block Analysis

The `%implements` directive is required by TLC block files and is used by the Target Language Compiler to verify the block type and language supported by the block. The `%function` directive starts a function declaration and shows the name of the function, `Outputs`, and the arguments passed to it, `block` and `system`. These are the relevant records from the *`model`*`.rtw` file for this instance of the block.

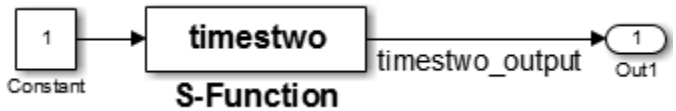
The last piece of the prototype is `Output`. This means that any line that is not a TLC directive is output by the function to the current file that is selected in TLC. So, nondirective lines in the `Outputs` function become generated code for the block.

The most complicated piece of this TLC block example is the `%roll` directive. TLC uses this directive to provide automatic generation of `for` loops, depending on input/output widths and whether the inputs are contiguous in memory. This example uses the typical form of accessing outputs and inputs from within the body of the roll, using `LibBlockOutputSignal` and `LibBlockInputSignal` to access the outputs and inputs and perform the multiplication and assignment. Note that this TLC file supports any valid signal width.

The only function used to implement this block is `Outputs`. For more complicated blocks, other functions are declared as well. You can find examples of more complicated inlining TLC files in *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/tlc_c` and *`matlabroot`*`/toolbox/simulink/blocks/tlc_c`, and by looking at the code for built-in blocks in *`matlabroot`*`/rtw/c/tlc/blocks`.

The timestwo Model

This simple model uses the `timestwo` S-function and shows the `MdlOutputs` function from the generated *`model`*`.c` file, which contains the inlined S-function code.



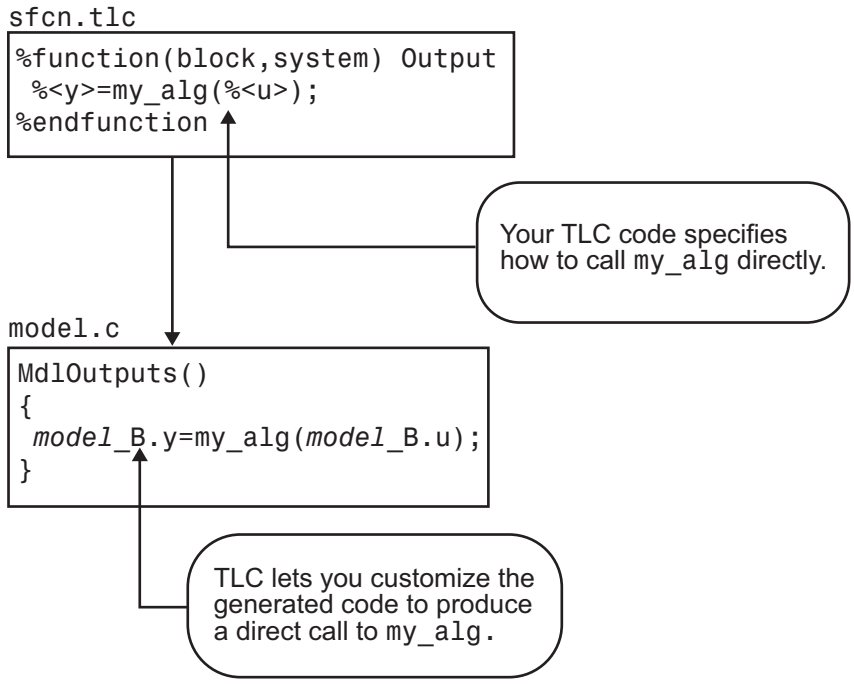
Model Outputs Code

```
/* Model output function */
static void timestwo_ex_output(int_T tid)
```

```
{

  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by two */
  timestwo_ex_B.timestwo_output = timestwo_ex_P.Constant_Value
  * 2.0;

  /* Outport: '<Root>/Out1' */
  timestwo_ex_Y.Out1 = timestwo_ex_B.timestwo_output;
}
```

Wrapper Inlined S-Function Example

The following diagram illustrates inlining an S-function as a wrapper. The algorithm is directly called from the generated model code, removing the S-function overhead but maintaining the user function.



This is the inlining TLC file for a wrapper version of the `timestwo` block.

```
%implements "timestwo" "C"

%% Function: BlockTypeSetup ===================================
%%
%function BlockTypeSetup(block, system) void
  %% Add function prototype to model's header file
  %<LibCacheFunctionPrototype...
   ("extern void mytimestwo(real_T* in,real_T* out,int_T els);")>
  %% Add file that contains "myfile" to list of files to be compiled
  %<LibAddToModelSources("myfile")>
%endfunction

%% Function: Outputs ==========================================
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign outPtr = LibBlockOutputSignalAddr(0, "", "", 0)
  %assign inPtr = LibBlockInputSignalAddr(0, "", "",0)
  %assign numEls = LibBlockOutputSignalWidth(0)
  /* Multiply input by two */
  mytimestwo(%<inPtr>,%<outPtr>,%<numEls>);

%endfunction
```

Analysis

The function `BlockTypeSetup` is called once for each type of block in a model; it doesn't produce output directly like the `Outputs` function. Use `BlockTypeSetup` to include a function prototype in the *`model`*`.h` file and to tell the build process to compile an additional file, `myfile.c`.

Instead of performing the multiplication directly, the `Outputs` function now calls the function `mytimestwo`. All instances of this block in the model call the same function to perform the multiplication. The resulting model function, `MdlOutputs`, then becomes

```
static void timestwo_ex_output(int_T tid)
{
  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by two */
  mytimestwo(&model_B.Constant_Value,&model_B.S_Function,1);

  /* Outport Block: <Root>/Out1 */
model_Y.Out1 = model_B.S_Function;
}
```

Target Language Compiler Tutorials

Advice About TLC Tutorials

The fastest and easiest way to understand the Target Language Compiler (TLC) is to run it, paying attention to how TLC scripts transform compiled Simulink models (*model*`.rtw` files) into source code. The tutorials in this chapter are designed to highlight the principal reasons for and techniques of using TLC. The tutorials provide a number of TLC exercises, each one organized as a major section.

The example models, S-functions, and TLC files for the exercises are located in *matlabroot*`/toolbox/rtw/rtwdemos/tlctutorial`, where *matlabroot* is the MATLAB root folder on your system. In this chapter, this folder is referred to as `tlctutorial`. Each example is located in a separate subfolder within `tlctutorial`. Within that subfolder, you can find solutions to the problem in a `solutions` subfolder.

Note: Before you begin the tutorial, copy the entire `tlctutorial` folder to a local working folder. The files are together, and if you make mistakes or want fresh examples to try again, you can recopy files from the original `tlctutorial` folder.

Each tutorial exercise is limited in scope, requiring just a small amount of experimentation. The tutorial explains details about TLC that will help customize and optimize code for Simulink Coder projects.

Note: You should not customize TLC files in the folder *matlabroot*`/rtw/c/tlc` even though the capability exists to do so. Such TLC customizations might not be applied during the code generation process and can lead to unpredictable results.

The tutorials progress in difficulty from basic to more advanced. To get the most out of them, you should be familiar with

- Working in the MATLAB environment
- Building Simulink models
- Using Simulink Coder software to generate code for target systems
- High-level language concepts (for example, C or Fortran programming)

If you encounter terms in the tutorials that you do not understand, it may be helpful to read “Code Generation Concepts” to acquaint yourself with the basic goals and methods

of TLC programming. Similarly, if you see TLC keywords, built-in functions, or directives that you would like to know more about, see “Directives and Built-In Functions”. Descriptions of TLC library functions are provided in “TLC Function Library Reference”.

The examples used in the tutorial are:

Example	Description
guide	Illustrative record file
timesN	An example C file S-function for multiplying an input by N
tlcdebug	An example using TLC Debugger
wrapper	Example TLC file for S-function `wrapsfcn.c`

Read Record Files with TLC

In this section...

Tutorial Overview

Objective: Understand the structure of record files and learn how to parse them with TLC directives.

Folder: *`matlabroot`*`/toolbox/rtw/rtwdemos/tlctutorial/guide`

In this tutorial you interpret a simple file of structured records with a series of TLC scripts. You will learn how records are structured, and how TLC `%assign` and `%<>` *token expansion* directives are used to process them. In addition, the tutorial illustrates loops using `%foreach`, and scoping using `%with`.

The tutorial includes these steps, which you should follow sequentially:

1. **Structure of Record Files** — Some background and a simple example
2. **Interpret Records** — Presenting contents of the record file
3. **Anatomy of a TLC Script** — Deconstructing the presentation
4. **Modify `read-guide.tlc`** — Experiment with TLC
5. **Pass and Use a Parameter**— Pass parameters from the command line to TLC files
6. **Review**

Structure of Record Files

The Simulink Coder code generator compiles models into a structured form called a record file, referred to as *`model`*`.rtw`. Such compiled model files are similar in syntax

and organization to source model files, in that they contain a series of hierarchically nested records of the form

```
recordName {itemName itemValue}
```

Item names are alphabetic. Item values can be strings or numbers. Numeric values can be scalars, vectors, or matrices. Curly braces set off the contents of each record, which may contain one or more items, delimited by space, tab, or return characters.

In a *`model`*`.rtw` file, the top-level (first) record's name is `CompiledModel`. Each block is represented by a subrecord within it, identified by the block's name. TLC can parse well-formed record files, as this exercise illustrates.

The following listing is a valid record file that TLC can parse, although not one for which it can generate code. Comments are indicated by a pound sign (#):

```
#
# File: guide.rtw Illustrative record file, which can't be used by Simulink
#                                   Note: string values MUST be in quotes
Top {                              # Outermost Record, called Top
  Date         "21-Aug-2008"       # Name/Value pair named Top.Date
  Employee {                       # Nested record within the Top record
    FirstName  "Arthur"            #   Alpha field Top.Employee.FirstName
    LastName   "Dent"              #   Alpha field Top.Employee.LastName
    Overhead    1.78               #   Numeric field Top.Employee.Overhead
    PayRate     11.50              #   Numeric field Top.Employee.PayRate
    GrossRate   0.0                #   Numeric Field Top.Employee.GrossRate
  }                                # End of Employee record
  NumProject   3                   # Indicates length of following list
  Project {                        # First list item, called Top.Project[0]
    Name       "Tea"               #   Alpha field Name, Top.Project[0].Name
    Difficulty 3                   #   Numeric field Top.Project[0].Difficulty
  }                                # End of first list item
  Project {                        # Second list item, called Top.Project[1]
    Name       "Gillian"           #   Alpha field Name, Top.Project[1].Name
    Difficulty 8                   #   Numeric field Top.Project[1].Difficulty
  }                                # End of second list item
  Project {                        # Third list item, called Top.Project[2]
    Name       "Zaphod"            #   Alpha field Name, Top.Project[2].Name
    Difficulty 10                  #   Numeric field Top.Project[2].Difficulty
  }                                # End of third list item
}                                  # End of Top record and of file
```

As long as programmers know the names of records and fields, and their expected contents, they can compose TLC statements to read, parse, and manipulate record file data.

Interpret Records

Here is the output from a TLC program script that reads `guide.rtw`, interprets its records, manipulates field data, and formats descriptions, which are directed to the MATLAB Command Window:

```
Using TLC you can:
* Directly access a field's value, e.g.
  %<Top.Date> -- evaluates to:
  "21-Aug-2008"

* Assign contents of a field to a variable, e.g.
  "%assign worker = Top.Employee.FirstName"
  worker expands to Top.Employee.FirstName = Arthur

* Concatenate string values, e.g.
  "%assign worker = worker + " " + Top.Employee.LastName"
  worker expands to worker + " " + Top.Employee.LastName = "Arthur Dent"

* Perform arithmetic operations, e.g.
  "%assign wageCost = Top.Employee.PayRate * Top.Employee.Overhead"
  wageCost expands to Top.Employee.PayRate * Top.Employee.Overhead <- 11.5 * 1.78 = 20.47

* Put variables into a field, e.g.
  Top.Employee.GrossRate starts at 0.0
  "%assign Top.Employee.GrossRate = wageCost"
  Top.Employee.GrossRate expands to wageCost = 20.47

* Index lists of values, e.g.
  "%assign projects = Top.Project[0].Name + ", " + Top.Project[1].Name..."
  "+ ", " + Top.Project[2].Name"
  projects expands to Top.Project[0].Name + ", " + Top.Project[1].Name
  + ", " + Top.Project[2].Name = Tea, Gillian, Zaphod

* Traverse and manipulate list data via loops, e.g.
  - At top of Loop, Project = Tea; Difficulty = 3
  - Bottom of Loop, i = 0; diffSum = 3.0
  - At top of Loop, Project = Gillian; Difficulty = 8
  - Bottom of Loop, i = 1; diffSum = 11.0
  - At top of Loop, Project = Zaphod; Difficulty = 10
  - Bottom of Loop, i = 2; diffSum = 21.0
  Average Project Difficulty expands to diffSum / Top.NumProject = 21.0 / 3 = 7.0
```

This output from `guide.rtw` was produced by invoking TLC from the MATLAB Command Window, executing a script called `read-guide.tlc`. Do this yourself now, by following these steps:

1. In MATLAB, change folder (`cd`) to your copy of `tlctutorial/guide` within your working folder.
2. To produce the output just listed, process `guide.rtw` with the TLC script `read-guide.tlc` by typing the following command:

   ```
   tlc -v -r guide.rtw read-guide.tlc
   ```

Note command usage:

- The `-r` switch (for read) identifies the input data file, in this case `guide.rtw`.
- The TLC script handling the data file is specified by the last token typed.
- The `-v` switch (for verbose) directs output to the command window, unless the TLC file handles this itself.

Anatomy of a TLC Script

You now dissect the script you just ran. Each "paragraph" of output from `guide.tlc` is discussed in sequence in the following brief sections:

Coding Conventions

These are some basic TLC syntax and coding conventions:

`%% Comment`	TLC comment, which is not output
`/* comment */`	Comment, to be output
`%keyword`	TLC directive (keyword), start with "%"
`%<expr>`	TLC token operator
`.` (period)	Scoping operator, for example, `Top.Lev2.Lev3`
`...` (at end-of-line)	Statement continuation (line break is not output)
\ (at end-of-line)	Statement continuation (line break is output)
`localvarIdentifier`	Local variables start in lowercase

`GlobalvarIdentifier`	Global variables start in uppercase
`RecordIdentifier`	Record identifiers start in uppercase
`EXISTS()`	TLC built-in functions are named in uppercase **Note:** TLC identifiers are case-sensitive.

For further information, see "TLC Coding Conventions" on page 8-23.

File Header

The file `read-guide.tlc` begins with:

```
%% File: read-guide.tlc   (This line is a TLC Comment, and will not print)
%%
%% To execute this file, type: tlc -v -r guide.rtw read-guide.tlc
%% Set format for displaying real values (default is "EXPONENTIAL")
%realformat "CONCISE"
```

- Lines 1 through 4 — Text on a line following the characters `%%` is treated as a comment (ignored, not interpreted or output).
- Line 5 — As explained in the text of the fourth line, is the TLC directive (keyword) `%realformat`, which controls how subsequent floating-point numbers are formatted when displayed in output. Here we want to minimize the digits displayed.

Token Expansion

The first section of output is produced by the script lines:

```
Using TLC you can:
* Directly access a field's value, e.g.
%assign td = "%" + "<Top.Date>"
  %<td> -- evaluates to:
  "%<Top.Date>"
```

- Lines 1 and 2 — (and lines that contain no TLC directives or tokens) are simply echoed to the output stream, including leading and trailing spaces.
- Line 3 — Creates a variable named `td` and assigns the string value `%<Top.Date>` to it. The `%assign` directive creates new and modifies existing variables. Its general syntax is:

  ```
  %assign ::variable = expression
  ```
  The optional double colon prefix specifies that the variable being assigned to is a global variable. In its absence, TLC creates or modifies a local variable in the current scope.

- Line 4 — Displays

  ```
  %<Top.Date> -- evaluates to:
  ```
  The preceding line enables TLC to print `%<Top.Date>` without expanding it. It constructs the string by pasting together two literals.

  ```
  %assign td = "%" + "<Top.Date>"
  ```
  As discussed in "String Processing Plus" on page 3-10, the plus operator concatenates strings as and adds numbers, vectors, matrices, and records.
- Line 5 — Evaluates (expands) the record `Top.Date`. More precisely, it evaluates the field `Date` which exists in scope `Top`. The syntax `%<expr>` causes expression `expr` (which can be a record, a variable, or a function) to be evaluated. This operation is sometimes referred to as an *eval*.

Note: You cannot nest the `%<expr>` operator (that is, `%<foo%<bar>>` is not allowed).

Note: When you use the `%<expr>` operator within quotation marks, for example, `"%<Top.Date>"`, TLC expands the expression and then encloses the result in quotation marks. However, placing `%assign` within quotation marks, for example, `"%assign foo = 3"`, simply echoes the statement enclosed in quotation marks to the output stream. No assignment results (the value of `foo` remains unchanged or undefined).

General Assignment

The second section of output is produced by the script lines:

```
* Assign contents of a field to a variable, e.g.
%assign worker = Top.Employee.FirstName
  "%assign worker = Top.Employee.FirstName"
  worker expands to Top.Employee.FirstName = %<worker>
```

- Line 1 — Echoed to output.
- Line 2 — An assignment of field `FirstName` in the `Top.Employee` record scope to a new local variable called `worker`.
- Line 3 — Repeats the previous statement, producing output by enclosing it in quotation marks.
- Line 4 — Explains the following assignment and illustrates the token expansion. The token `%<worker>` expands to `Arthur`.

String Processing Plus

The next section of the script illustrates string concatenation, one of the uses of the "+" operator:

```
* Concatenate string values, e.g.
%assign worker = worker + " " + Top.Employee.LastName
  "%assign worker = worker + " " + Top.Employee.LastName"
  worker expands to worker + " " + Top.Employee.LastName = "%<worker>"
```

- Line 1 — Echoed to output.
- Line 2 — Performs the concatenation.
- Line 3 — Echoes line 2 to the output.
- Line 4 — Describes the operation, in which a variable is concatenated to a field separated by a space character. An alternative way to do this, without using the + operator, is

  ```
  %assign worker = "%<Top.Employee.FirstName> %<Top.Employee.LastName>"
  ```

  The alternative method uses evals of fields and is equally efficient.

The + operator, which is associative, also works for numeric types, vectors, matrices, and records:

- Numeric Types — Add two expressions together; both operands must be numeric. For example:

  ```
  * Numeric Type example, e.g.
    Top.Employee.PayRate = %<Top.Employee.PayRate>
    Top.Employee.Overhead = %<Top.Employee.Overhead>
  %assign td = Top.Employee.PayRate + Top.Employee.GrossRate
    td = Top.Employee.PayRate + Top.Employee.Overhead
    td evaluates to %<td>
  ```

  Output:

  ```
  * Numeric Type example, e.g.
    Top.Employee.PayRate = 11.5
    Top.Employee.Overhead = 1.78
    td = Top.Employee.PayRate + Top.Employee.Overhead
    td evaluates to 13.28
  ```

- Vectors — If the first argument is a vector and the second is a scalar value, TLC appends the scalar value to the vector. For example:

```
* Vector example, e.g.
%assign v1 = [0, 1, 2, 3]
  v1 is %<v1>
%assign tp1d = Top.Project[1].Difficulty
  Top.Project[1].Difficulty is %<tp1d>
%assign v2 = v1 + tp1d
  v2 = v1 + Top.Project[1].Difficulty
  v2 evaluates to: %<v2>
```

Output:

```
* Vector example, e.g.
  v1 is [0, 1, 2, 3]
  Top.Project[1].Difficulty is 8
  v2 = v1 + Top.Project[1].Difficulty
  v2 evaluates to: [0, 1, 2, 3, 8]
```

- Matrices — If the first argument is a matrix and the second is a vector of the same column-width as the matrix, TLC appends the vector as another row to the matrix. For example:

```
* Matrices example, e.g.
%assign mx1 = [[4, 5, 6, 7]; [8, 9, 10, 11]]
  mx1 is %<mx1>
  v1 is %<v1>
%assign mx = mx1 + v1
  mx = mx1 + v1
  mx evaluates to %<mx>
```

Output:

```
* Matrices example, e.g.
  mx1 is [ [4, 5, 6, 7]; [8, 9, 10, 11] ]
  v1 is [0, 1, 2, 3]
  mx = mx1 + v1
  mx evaluates to [ [4, 5, 6, 7]; [8, 9, 10, 11]; [0, 1, 2, 3] ]
```

- Records — If the first argument is a record, TLC adds the second argument as a parameter identifier (with its current value). For example:

```
* Record example, e.g.
%assign StartDate = "August 28, 2008"
  StartDate is %<StartDate>
%assign tsd = Top + StartDate
  Top + StartDate
  Top.StartDate evaluates to %<Top.StartDate>
```

Output:

```
  * Record example, e.g.
    StartDate is August 28, 2008
    Top + StartDate
    Top.StartDate evaluates to August 28, 2008
```

Arithmetic Operations

TLC provides a full complement of arithmetic operators for numeric data. In the next portion of our TLC script, two numeric fields are multiplied:

```
* Perform arithmetic operations, e.g.
%assign wageCost = Top.Employee.PayRate * Top.Employee.Overhead
  "%assign wageCost = Top.Employee.PayRate * Top.Employee.Overhead"
  wageCost expands to Top.Employee.PayRate * Top.Employee.Overhead ...
 <- %<Top.Employee.PayRate> * %<Top.Employee.Overhead> = %<wageCost>
```

- Line 1 — Echoed to output.
- Line 2 — `%assign` statement that computes the value, which TLC stores in local variable `wageCost`.
- Line 3 — Echoes the operation in line 2.
- Lines 4 and 5 — Compose a single statement. The ellipsis (typed as three consecutive periods, for example, `...`) signals that a statement is continued on the following line, but if the statement has output, TLC does not insert a line break. To continue a statement and insert a line break, replace the ellipsis with a backslash (\).

Modify Records

Once read into memory, you can modify and manipulate records just like variables you create by assignment. The next segment of `read-guide.tlc` replaces the value of record field `Top.Employee.GrossRate`:

```
* Put variables into a field, e.g.
%assign Top.Employee.GrossRate = wageCost
  "%assign Top.Employee.GrossRate = wageCost"
  Top.Employee.GrossRate expands to wageCost = %<Top.Employee.GrossRate>
```

Such changes to records are nonpersistent (because record files are inputs to TLC; other file types, such as C source code, are outputs), but can be useful.

You can use several TLC directives besides `%assign` to modify records:

`%createrecord`	Creates new top-level records, and might also specify subrecords within them, including name/value pairs.

`%addtorecord`	Adds fields to an existing record. The new fields can be name/value pairs or aliases to existing records.
`%mergerecord`	Combines one or more records. The first record contains itself plus copies of the other records' contents specified by the command, in sequence.
`%copyrecord`	Creates a new record as `%createrecord` does, except the components of the record come from the existing record you specify.
`%undef var`	Removes (deletes) `var` (a variable or a record) from scope. If `var` is a field in a record, TLC removes the field from the record. If `var` is a record array (list), TLC removes the first element of the array; the remaining elements remain accessible. You can remove only records you create with `%createrecord` or `%copyrecord`.

See "Target Language Compiler Directives" for details on these directives.

Index Lists

Record files can contain lists, or sequences of records having the same identifier. Our example contains a list of three records identified as `Project` within the `Top` scope. List references are indexed, numbered from 0, in the order in which they appear in the record file. Here is TLC code that compiles data from the `Name` field of the `Project` list:

```
* Index lists of values, e.g.
%assign projects = Top.Project[0].Name + ", " + Top.Project[1].Name...
+ ", " + Top.Project[2].Name
  "%assign projects = Top.Project[0].Name + ", " + Top.Project[1].Name..."
  "+ ", " + Top.Project[2].Name"
  projects expands to Top.Project[0].Name + ", " + Top.Project[1].Name
  + ", " + Top.Project[2].Name = %<projects>
```

The `Scope.Record[n].Field` syntax is similar to that used in C to reference elements in an array of structures.

While explicit indexing, such as the above, is perfectly acceptable, it is often preferable to use a loop construct when traversing entire lists, as shown in "Loop Over Lists" on page 3-13.

Loop Over Lists

By convention, the section of a record file that a list occupies is preceded by a record that indicates how many list elements are present. In *`model`*`.rtw` files, such parameters are

declared as `Num`*`Ident`*, where *`Ident`* is the identifier used for records in the list that follows. In `guide.rtw`, the `Project` list looks like this:

```
  NumProject    3                   # Indicates length of following list
  Project {                         # First list item, called Top.Project[0]
    Name        "Tea"               #   Alpha field Name, Top.Project[0].Name
    Difficulty  3                   #   Numeric field Top.Project[0].Difficulty
  }                                 # End of first list item
  Project {                         # Second list item, called Top.Project[1]
    Name        "Gillian"           #   Alpha field Name, Top.Project[1].Name
    Difficulty  8                   #   Numeric field Top.Project[1].Difficulty
  }                                 # End of second list item
  Project {                         # Third list item, called Top.Project[2]
    Name        "Zaphod"            #   Alpha field Name, Top.Project[2].Name
    Difficulty  10                  #   Numeric field Top.Project[2].Difficulty
  }                                 # End of third list item
```

Thus, the value of `NumProject` describes how many `Project` records occur.

Note: `model.rtw` files might also contain records that start with `Num` but are not list-size parameters. TLC does not require that list size parameters start with `Num`. Therefore you need to be cautious when interpreting `Num`*`Ident`* record identifiers. The built-in TLC function `SIZE()` can determine the number of records in a specified scope, hence the length of a list.

The last segment of `read-guide.tlc` uses a `%foreach` loop, controlled by the `NumProject` parameter, to iterate the `Project` list and manipulate its values.

```
* Traverse and manipulate list data via loops, e.g.
%assign diffSum = 0.0
%foreach i = Top.NumProject
  - At top of Loop, Project = %<Top.Project[i].Name>; Difficulty =...
 %<Top.Project[i].Difficulty>
  %assign diffSum = diffSum + Top.Project[i].Difficulty
  - Bottom of Loop, i = %<i>; diffSum = %<diffSum>
%endforeach
%assign avgDiff = diffSum / Top.NumProject
  Average Project Difficulty expands to diffSum / Top.NumProject = %<diffSum> ...
 / %<Top.NumProject> = %<avgDiff>
```

As you may recall, the TLC output looks like this:

```
* Traverse and manipulate list data via loops, e.g.
- At top of Loop, Project = Tea; Difficulty = 3
  - Bottom of Loop, i = 0; diffSum = 3.0
  - At top of Loop, Project = Gillian; Difficulty = 8
  - Bottom of Loop, i = 1; diffSum = 11.0
  - At top of Loop, Project = Zaphod; Difficulty = 10
```

```
  - Bottom of Loop, i = 2; diffSum = 21.0
Average Project Difficulty expands to diffSum / Top.NumProjects = 21.0 / 3 = 7.0
```

After initializing the summation variable `diffSum`, a `%foreach` loop is entered, with variable `i` declared as the loop counter, iterating up to `NumProject`. The scope of the loop is all statements encountered until the corresponding `%endforeach` is reached (`%foreach` loops may be nested).

Note: Loop iterations implicitly start at zero and range to one less than the index that specifies the upper bound. The loop index is local to the loop body.

Modify `read-guide.tlc`

Now that you have studied `read-guide.tlc`, it is time to modify it. This exercise introduces two important TLC facilities, *file control* and *scoping control*. You implement both within the `read-guide.tlc` script.

File Control Basics

TLC scripts almost invariably produce output in the form of streams of characters. Output is normally directed to one or more buffers and files, collectively called *streams*. So far, you have directed output from `read-guide.tlc` to the MATLAB Command Window because you included the `-v` switch on the command line. Prove this by omitting `-v` when you run `read-guide.tlc`. Type

```
tlc -r guide.rtw read-guide.tlc
```

Nothing appears to happen. In fact, the script was executed, but output was directed to a null device (sometimes called the “bit bucket”).

There is one active output file, even if it is null. To specify, open, and close files, use the following TLC directives:

```
%openfile streamid ="filename" , "mode"
%closefile streamid
%selectfile streamid
```

If you do not give a filename, subsequent output flows to the memory buffer named by `streamid`. If you do not specify a mode, TLC opens the file for writing and deletes any existing content (subject to system-level file protection mechanisms). Valid mode identifiers are `a` (append) and `w` (write, the default). Enclose these characters in quotes.

The `%openfile` directive creates a file/buffer (in `w` mode), or opens an existing one (in `a` mode). Note the required equals sign for file specification. Multiple streams can be open for writing, but only one can be active at one time. To switch output streams, use the `%selectfile` directive. You do not need to close files until you are done with them.

The default output stream, which you can respecify with the stream ID `NULL_FILE`, is `null`. Another built-in stream is `STDOUT`. When activated using `%selectfile`, `STDOUT` directs output to the MATLAB Command Window.

Note: The streams `NULL_FILE` and `STDOUT` are always open. Specifying them with `%openfile` generates errors. Use `%selectfile` to activate them.

The directive `%closefile` closes the current output file or buffer. Until an `%openfile` or a `%selectfile` directive is encountered, output goes to the previously opened stream (or, if none exists, to null). Use `%selectfile` to designate an open stream for reading or writing. In practice, many TLC scripts write pieces of output data to separate buffers, which are then selected in a sequence and their contents spooled to one or more files.

Implement Output File Control

In your `tlctutorial/guide` folder, find the file `read-guide-file-src.tlc`. The supplied version of this file contains comments and three lines of text added. Edit this file to implement output file control, as follows:

1. Open `read-guide-file-src.tlc` in your text editor.
2. Save the file as `read-guide-file.tlc`.
3. Note five comment lines that begin with `%% ->`.

   Under each of these comments, insert a TLC directive as indicated.
4. Save the edited file as `read-guide-file.tlc`.
5. Execute `read-guide-file.tlc` with the following command:

   ```
   tlc -r guide.rtw read-guide-file.tlc
   ```

   If you succeeded, TLC creates the file `guidetext.txt` which contains the expected output, and the MATLAB Command Window displays

   ```
   *** Output being directed to file: guidetext.txt
   *** We're almost done . . .
   ```

```
*** Processing completed.
```

If you did not see these messages, or if a text file was not produced, review the material and try again. If problems persist, inspect `read-guide-file.tlc` in the `guide/solutions` subfolder to see how you should specify file control.

Scope Basics

"Structure of Record Files" on page 3-4 explains the hierarchical organization of records. Each record exists within a scope defined by the records in which it is nested. The example file, `guide.rtw`, contains the following scopes:

```
Top
Top.Employee
Top.Project[0]
Top.Project[1]
Top.Project[2]
```

To refer to a field or a record, specify its scoping, even if no other context that contains the identifier exists. For example, in `guide.rtw`, the field `FirstName` exists only in the scope `Top.Employee`. You must refer to it as `Top.Employee.FirstName` whenever accessing it.

When models present scopes that are deeply nested, this can lead to extremely long identifiers that are tedious and error prone to type. For example:

```
CompiledModel.BlockOutputs.BlockOutput.ReusedBlockOutput
```

This identifier has a scope that is long and has similar item names that you could easily enter incorrectly.

The `%with/%endwith` directive eases the burden of coding TLC scripts and clarifies their flow of control. The syntax is

```
%with RecordName
  [TLC statements]
%endwith
```

Every `%with` is eventually followed by an `%endwith`, and these pairs might be nested (but not overlapping). If `RecordName` is below the top level, you need not include the top-level scope in its description. For example, to make the current scope of `guide.rtw` `Top.Employee`, you can specify

```
%with Employee
```

```
  [TLC statements]
%endwith
```

Naturally, `%with Top.Employee` is also valid syntax. Once bracketed by `%with`/`%endwith`, record identifiers in TLC statements do not require you to specify their outer scope. However, note the following conditions :

- You can access records outside of the current `%with` scope, but you must qualify them fully (for example, using record name and fields).
- Whenever you make assignments to records inside a `%with` directive, you must qualify them fully.

Change Scope Using `%with`

In the last segment of this exercise, you modify the TLC script by adding a `%with`/`%endwith` directive. You also need to edit record identifier names (but not those of local variables) to account for the changes of scope resulting from the `%with` directives.

1. Open the TLC script `read-guide-scope-src.tlc` in the text editor.
2. Save the file as `read-guide-scope.tlc`.
3. Note comment lines that commence with `%% ->`.

   Under each of these comments, insert a TLC directive or modify statements already present, as indicated.
4. Save the edited file as `read-guide-scope.tlc`.
5. Execute `read-guide-scope.tlc` with the following command:

   ```
   tlc -v -r guide.rtw read-guide-scope.tlc
   ```

The output should be exactly the same as from `read-guide.tlc`, except possibly for white space that you might have introduced by indenting sections of code inside `%with`/`%endwith` or by eliminating blank lines.

Fully specifying a scope inside a `%with` context is not an error, it is simply unnecessary. However, failing to fully specify its scope when assigning it to a record (for example, `%assign GrossRate = wageCost`) is invalid.

If errors result from running the script, review the discussion of scoping above and edit `read-guide-scope.tlc` to eliminate them. As a last resort, inspect `read-guide-scope.tlc` in the `/solutions` subfolder to see how you should have handled scoping in this exercise.

For additional information, see "Scopes in the *model*.rtw File" on page 5-4 and "Variable Scoping" on page 6-50.

Pass and Use a Parameter

You can use the TLC commands and built-in functions to pass parameters from the command line to the TLC file being executed. The most general command switch is `-a`, which assigns arbitrary variables. For example:

```
tlc -r input.rtw -avar=1 -afoo="abc" vars.tlc
```

The result of passing this pair of strings via `-a` is the same as declaring and initializing local variables in the file being executed (here, `vars.tlc`). For example:

```
%assign var = 1
%assign foo = "abc"
```

You do not need to declare such variables in the TLC file, and they are available for use when set with `-a`. However, errors result if the code assigns undeclared variables that you do not specify with the `-a` switch when invoking the file. Also note that (in contrast to the `-r` switch) a space should not separate `-a` from the parameter you are declaring.

In the final section of this tutorial, you use the built-in function `GET_COMMAND_SWITCH()` to print the name of the record file being used in the TLC script, and provide a parameter to control whether or not the code is suppressed. By default the code is executed, but is suppressed if the command line contains `-alist=0`:

1. Open the TLC script `read-guide-param-src.tlc` in your text editor.
2. Save the file as `read-guide-param.tlc`.
3. To enable your program to access the input filename from the command line, do the following:

   a. Below the line `%selectfile STDOUT`, add the line:

      ```
      %assign inputfile = GET_COMMAND_SWITCH ("r")
      ```

      The `%assign` directive declares and sets variables. In this instance, it holds a string filename identifier. `GET_COMMAND_SWITCH()` returns whatever string argument follows a specified TLC command switch. You must use UPPERCASE for built-in function names.

   b. Change the line "`*** WORKING WITH RECORDFILE`" to read as follows:

```
*** WORKING WITH RECORDFILE %<inputfile>
```

4 To control whether or not a section of TLC code is executed, do the following:

a Below the line "`%assign inputfile = GET_COMMAND_SWITCH ("r")`", add:

```
%if (!EXISTS(list))
 %assign list = 1
%endif
```

The program checks whether a list parameter has been declared, via the intrinsic (built-in) function `EXISTS()`. If no list variable exists, the program assigns one. This defines `list` and by default its value is `TRUE`.

b Enclose lines of code within an `%if` block.

```
%if (list)
    * Assign contents of a field to a variable, e.g.
      %assign worker = FirstName
      "%assign worker = FirstName"
      worker expands to FirstName = %<worker>
%endif
```

Now the code to assign `worker` is sent to the output only when `list` is `TRUE`.

c Save `read-guide-param.tlc`.

5 Execute `read-guide-param.tlc` and examine the output, using the command

```
tlc -r guide.rtw read-guide-param.tlc
```

This yields

```
*** WORKING WITH RECORDFILE [guide.rtw]
* Assign contents of a field to a variable, e.g.
      "%assign worker = FirstName"
      worker expands to FirstName = Arthur
***END
```

6 Execute`read-guide-param.tlc` with the command:

```
tlc -r guide.rtw -alist=0 read-guide-param.tlc
```

With the `-alist=0` switch, the output displays only the information outside of the `if` statement.

```
*** WORKING WITH RECORDFILE [guide.rtw]
***END
```

Review

The preceding exercises examined the structure of record files, and expanded on how to use TLC directives. The following TLC directives are commonly used in TLC scripts (see "Target Language Compiler Directives" for detailed descriptions):

`%addincludepath`	Enable TLC to find included files.
`%addtorecord`	Add fields to existing record. New fields can be name/value pairs or aliases to existing records.
`%assign`	Create or modify variables.
`%copyrecord`	Create new recordand, if applicable, specify subrecords within them, including name/value pairs. The components of the record come from the existing record specified.
`%createrecord`	Create new top-level records and, if applicable, specify subrecords within them, including name/value pairs.
`%foreach/%endforeach`	Iterate loop variable from 0 to upper limit.
`%if/%endif`	Control whether code is executed, as in C.
`%include`	Insert one file into another, as in C.
`%mergerecord`	Combine one or more records. The first record contains itself plus copies of the other records contents specified by the command, in sequence.
`%selectfile`	Direct outputs to a stream or file.
`%undef var`	Remove (delete) `var` (a variable or a record) from scope. If `var` is a field in a record, TLC removes the field from the record. If `var` is a record array (list), TLC removes the first element of the array; the remaining elements remain accessible. Only records created via `%createrecord` or `%copyrecord` can be removed.
`%with/%endwith`	Add scope to simplify referencing blocks.

In "Pass and Use a Parameter" on page 3-19, you used TLC built in functions. See "Target Language Compiler Directives" for more information.

Inline S-Functions with TLC

In this section...

timesN Tutorial Overview

Objective: To understand how TLC works with an S-function.

Folder: *matlabroot*/toolbox/rtw/rtwdemos/tlctutorial/timesN

In this tutorial, you generate versions of C code for existing S-function `timesN`.

The tutorial includes these steps:

1. **Noninlined Code Generation** — Via SimStructs and generic API
2. **Why Use TLC to Inline S-Functions?** — Benefits of inlining
3. **Create an Inlined S-Function** — Via custom TLC code

A later tutorial provides information and practice with "wrapping" S-functions.

Noninlined Code Generation

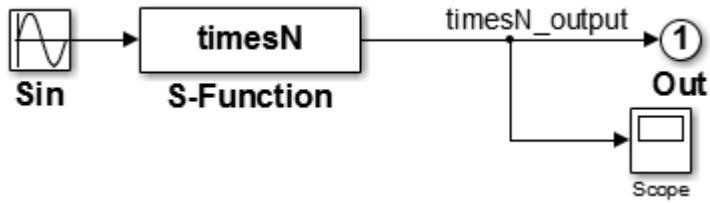
The tutorial folder `tlctutorial/timesN` in your working folder contains Simulink S-function `timesN.c`.

In this exercise, you generate noninlined code from the model `sfun_xN`.

1. Find the file `rename_timesN.tlc` in `tlctutorial/timesN`. Rename this file to `timesN.tlc`. This allows you to generate code.
2. In the MATLAB Command Window, create a MEX-file for the S-function:

   ```
   mex timesN.c
   ```

   This avoids picking up the version shipped with Simulink.
3. Open the model `sfun_xN`, which uses the `timesN` S-function. The block diagram looks like this.



4 Open the Configuration Parameters dialog box and select the **Solver** pane.

5 Set **Stop time** to `10.0`.

6 Set the **Solver Options**.

- **Type** to `Fixed-step`
- **Solver** to `Discrete (no continuous states)`
- **Fixed-step size** to `0.01`

7 Select the **Optimization** > **Signals and Parameters** pane, and make sure that **Inline parameters** is unchecked.

8 Select the **Code Generation** > **Comments** pane, and notice that **Include comments** is checked by default.

9 Select the **Code Generation** pane and check **Generate code only**.

The text of the **Build** button changes to **Generate Code**. Click **Apply**.

10 Click **Generate Code** to generate C code for the model.

11 Open the resulting file `sfun_xN_grt_rtw/sfun_xN.c` and view the `sfun_xN_output` portion, shown below.

```
/* Model output function */
static void sfun_xN_output(int_T tid)
{
  /* Sin: '<Root>/Sin' */
  sfun_xN_B.Sin = sin(sfun_xN_M->Timing.t[0] * sfun_xN_P.Sin_Freq +
                      sfun_xN_P.Sin_Phase) * sfun_xN_P.Sin_Amp +
    sfun_xN_P.Sin_Bias;

  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by 3.0 */
  sfun_xN_B.timesN_output = sfun_xN_B.Sin * 3.0;

  /* Outport: '<Root>/Out' */
  sfun_xN_Y.Out = sfun_xN_B.timesN_output;
  UNUSED_PARAMETER(tid);
}
```

Comments appear in the code because, in the **Code Generation > Comments** pane of the Configuration Parameters dialog box, **Include comments** is selected by default.

Why Use TLC to Inline S-Functions?

The Simulink Coder code generator includes a generic API that you can use to invoke user-written algorithms and drivers. The API includes a variety of callback functions — for initialization, output, derivatives, termination, and so on — as well as data structures. Once coded, these are instantiated in memory and invoked during execution via indirect function calls. Each invocation involves stack frames and other overhead that adds to execution time.

In a real-time environment, especially when many solution steps are involved, generic API calls can be unacceptably slow. The code generator can speed up S-functions in standalone applications that it generates by embedding user-written algorithms within auto-generated functions, rather than indirectly calling S-functions via the generic API. This form of optimization is called *inlining*. TLC inlines S-functions, resulting in faster, optimized code.

You should understand that TLC is not a substitute for writing C code S-functions. To invoke custom blocks within Simulink, you still have to write S-functions in C (or as MATLAB files), since simulations do not make use of TLC files. You can, however, prepare TLC files that inline specified S-functions to make your target code much more efficient.

Create an Inlined S-Function

TLC creates an *inlined S-function* whenever it detects a `.tlc` file with the same name as an S-function. Assuming the `.tlc` file has the expected formed, it directs construction of code that functionally duplicates the external S-function without incurring API overhead. See how this process works by completing the following steps:

1 If you have not done so already, find the file `rename_timesN.tlc` in `tlctutorial/timesN`. Rename this file to `timesN.tlc`, so you can use it to generate code. The executable portion of the file is

```
%implements "timesN" "C"

%% Function: Outputs ===========================================================
%%
%function Outputs(block, system) Output
  %assign gain =SFcnParamSettings.myGain
  /* %<Type> Block: %<Name> */
```

```
  %%
  /* Multiply input by %<gain> */
  %assign rollVars = ["U", "Y"]
  %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
    %<LibBlockOutputSignal(0, "", lcv, idx)> = \
    %<LibBlockInputSignal(0, "", lcv, idx)> * %<gain>;
  %endroll

%endfunction
```

2 Create the inline version of the S-function.

a On the **Optimization** > **Signals and Parameters** pane of the Configuration Parameters dialog box, check **Inline parameters** and click **Apply**.

b Change the diagram's label from `model: sfun_xN` to `model: sfun_xN_ilp`.

c Save the model as `sfun_x2_ilp`.

d In the **Code Generation** pane of the Configuration Parameters dialog box, click **Generate Code**. Source files are created in a new subfolder called `sfun_xN_ilp_grt_rtw`.

e Inspect the code in generated file `sfun_xN_ilp.c`:

```
/* Model output function */
static void sfun_xN_ilp_output(int_T tid)
{
  /* Sin: '<Root>/Sin' */
  sfun_xN_ilp_B.Sin = sin(sfun_xN_ilp_M->Timing.t[0]);

  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by 3.0 */
  sfun_xN_ilp_B.timesN_output = sfun_xN_ilp_B.Sin * 3.0;

  /* Outport: '<Root>/Out' */
  sfun_xN_ilp_Y.Out = sfun_xN_ilp_B.timesN_output;
  UNUSED_PARAMETER(tid);
}
```

Note: When the Simulink Coder software generates code and builds executables, it creates or uses a specific subfolder (called the build folder) to hold source, object, and make files. By default, the build folder is named *`model`*`_grt_rtw`.

Notice that checking the **Inline parameters** box did not change the code. This is because TLC inlines S-functions.

3 Continue the exercise by creating a standalone simulation.

a In the **Code Generation** pane of the Configuration Parameters dialog box, clear **Generate code only** and click **Apply**.

b In the **Data Import/Export** pane of the Configuration Parameters dialog box, under **Save to workspace**, check **Output**.

This specification causes the model's output data to be logged in your MATLAB workspace.

c In the **Code Generation** pane of the Configuration Parameters dialog box, click **Build** to generate code, compile, and link the model into an executable, named `sfun_xN_ilp.exe` (or, on UNIX® systems, `sfun_xN_ilp`).

d Confirm that the `timesN.tlc` file produces expected output by running the standalone executable. To run it, in the MATLAB Command Window, type

```
!sfun_xN_ilp
```

The following response appears:

```
** starting the model **
** created sfun_xN_ilp.mat **
```

e View or plot the contents of the `sfun_xN_ilp.mat` file to verify that the standalone model generated sine output ranging from -3 to +3. In the MATLAB Command Window, type

```
load sfun_xN_ilp.mat
plot (rt_yout)
```

Explore Variable Names and Loop Rolling

In this section...

`timesN` Looping Tutorial Overview

Objective: This example shows how you can influence looping behavior of generated code.

Folder: *`matlabroot`*`/toolbox/rtw/rtwdemos/tlctutorial/timesN`

Work with the model `sfun_xN` in `tlctutorial/timesN`. It has one source (a Sine Wave generator block), a times N gain block, an Out block, and a Scope block.

The tutorial guides you through following steps:

1. **Getting Started** — Set up the exercise and run the model
2. **Modify the Model** — Change the input width and see the results
3. **Change the Loop Rolling Threshold** — Change the threshold and see the results
4. **More About TLC Loop Rolling** — Parameterize loop behavior

Getting Started

1. Make `tlctutorial/timesN` your current folder, so that you can use the files provided.

   Note: You must use or create a working folder outside of *`matlabroot`* for Simulink models you make. You cannot build models in source folders.

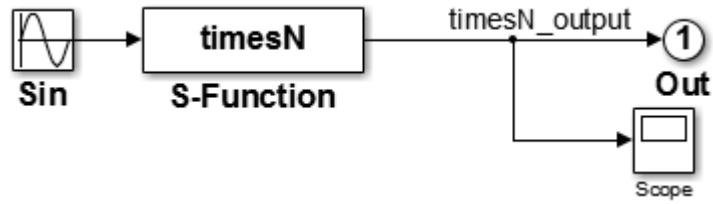
2. In the MATLAB Command Window, create a MEX-file for the S-function:

```
mex timesN.c
```

This avoids picking up the version shipped with Simulink.

Note: An error might occur if you have not previously run `mex -setup`.

3 Open the model file `sfun_xN`.



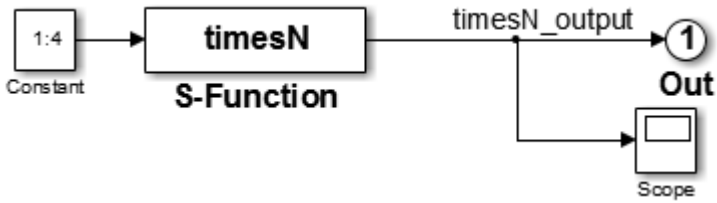
4 View the previously generated code in `sfun_xN_grt_rtw/sfun_xN.c`. Note that no loops exist in the code. This is because the input and output signals are scalar.

Modify the Model

1 Replace the Sine Wave block with a Constant block.

2 Set the parameter for the Constant block to 1:4, and change the top label, `model: sfun_xN`, to `model: sfun_vec`.

3 Save the edited model as `sfun_vec` (in `tlctutorial/timesN`). The model now looks like this.



4 Because the Constant block generates a vector of values, this is a vectorized model. Generate code for the model and view the `/*Model output function */` section of `sfun_vec.c` in your editor to observe how variables and `for` loops are handled. This function appears as follows:

```
/* Model output function */
static void sfun_vec_output(int_T tid)
{
  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by 3.0 */
  sfun_vec_B.timesN_output[0] = sfun_vec_P.Constant_Value[0] * 3.0;
  sfun_vec_B.timesN_output[1] = sfun_vec_P.Constant_Value[1] * 3.0;
  sfun_vec_B.timesN_output[2] = sfun_vec_P.Constant_Value[2] * 3.0;
  sfun_vec_B.timesN_output[3] = sfun_vec_P.Constant_Value[3] * 3.0;

  /* Outport: '<Root>/Out' */
  sfun_vec_Y.Out[0] = sfun_vec_B.timesN_output[0];
  sfun_vec_Y.Out[1] = sfun_vec_B.timesN_output[1];
  sfun_vec_Y.Out[2] = sfun_vec_B.timesN_output[2];
  sfun_vec_Y.Out[3] = sfun_vec_B.timesN_output[3];
  UNUSED_PARAMETER(tid);
}
```

Notice that there are four instances of the code that generates model outputs, corresponding to four iterations.

5. Set the parameter for the Constant block to 1:10, and save the model.
6. Generate code for the model and view the `/*Model output function */` section of `sfun_vec.c` in your editor to observe how variables and `for` loops are handled. This function appears as follows:

```
/* Model output function */
static void sfun_vec_output(int_T tid)
{
  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by 3.0 */
  {
    int_T i1;
    const real_T *u0 = &sfun_vec_P.Constant_Value[0];
    real_T *y0 = sfun_vec_B.timesN_output;
    for (i1=0; i1 < 10; i1++) {
      y0[i1] = u0[i1] * 3.0;
    }
  }

  {
    int32_T i;
    for (i = 0; i < 10; i++) {
      /* Outport: '<Root>/Out' */
      sfun_vec_Y.Out[i] = sfun_vec_B.timesN_output[i];
    }
  }

  UNUSED_PARAMETER(tid);
}
```

Notice that:

- The code that generates model outputs gets "rolled" into a loop. This occurs by default when the number of iterations exceeds 5.
- Loop index `i1` runs from 0 to 9.
- Pointer `*y0` is used and initialized to the output signal array.

Change the Loop Rolling Threshold

The code generator creates iterations or loops depending on the current value of the **Loop unrolling threshold** parameter.

The default value of **Loop unrolling threshold** is `5`. To change looping behavior for blocks in a model:

1 On the **Optimization** > **Signals and Parameters** pane of the Configuration Parameters dialog box, set **Loop unrolling threshold** to `12` and click `Apply`.

The parameter `RollThreshold` is now `12`. Loops will be generated only when the width of signals passing through a block exceeds 12.

Note: You cannot modify `RollThreshold` for specific blocks from the Configuration Parameters dialog box.

---

2 In the **Code Generation** pane of the Configuration Parameters dialog box, click **Generate Code** to regenerate the output.

3 Inspect `sfun_vec.c`. It will look like this:

```
/* Model output function */
static void sfun_vec_output(int_T tid)
{
  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by 3.0 */
  sfun_vec_B.timesN_output[0] = sfun_vec_P.Constant_Value[0] * 3.0;
  sfun_vec_B.timesN_output[1] = sfun_vec_P.Constant_Value[1] * 3.0;
  sfun_vec_B.timesN_output[2] = sfun_vec_P.Constant_Value[2] * 3.0;
  sfun_vec_B.timesN_output[3] = sfun_vec_P.Constant_Value[3] * 3.0;
  sfun_vec_B.timesN_output[4] = sfun_vec_P.Constant_Value[4] * 3.0;
  sfun_vec_B.timesN_output[5] = sfun_vec_P.Constant_Value[5] * 3.0;
  sfun_vec_B.timesN_output[6] = sfun_vec_P.Constant_Value[6] * 3.0;
  sfun_vec_B.timesN_output[7] = sfun_vec_P.Constant_Value[7] * 3.0;
  sfun_vec_B.timesN_output[8] = sfun_vec_P.Constant_Value[8] * 3.0;
  sfun_vec_B.timesN_output[9] = sfun_vec_P.Constant_Value[9] * 3.0;

  /* Outport: '<Root>/Out' */
  sfun_vec_Y.Out[0] = sfun_vec_B.timesN_output[0];
  sfun_vec_Y.Out[1] = sfun_vec_B.timesN_output[1];
  sfun_vec_Y.Out[2] = sfun_vec_B.timesN_output[2];
```

```
    sfun_vec_Y.Out[3] = sfun_vec_B.timesN_output[3];
    sfun_vec_Y.Out[4] = sfun_vec_B.timesN_output[4];
    sfun_vec_Y.Out[5] = sfun_vec_B.timesN_output[5];
    sfun_vec_Y.Out[6] = sfun_vec_B.timesN_output[6];
    sfun_vec_Y.Out[7] = sfun_vec_B.timesN_output[7];
    sfun_vec_Y.Out[8] = sfun_vec_B.timesN_output[8];
    sfun_vec_Y.Out[9] = sfun_vec_B.timesN_output[9];
    UNUSED_PARAMETER(tid);
  }
```

4 To activate loop rolling again, change the **Loop unrolling threshold** to 10 (or less) on the **Optimization > Signals and Parameters** pane.

Loop rolling is an important TLC capability for optimizing generated code. Take some time to study and explore its implications before generating code for production requirements.

More About TLC Loop Rolling

The following TLC `%roll` code is the `Outputs` function of `timesN.tlc`:

```
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %%
  /* Multiply input by %<gain> */
  %assign rollVars = ["U", "Y"]
  %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
     %<LibBlockOutputSignal(0, "", lcv, idx)> = \
    %<LibBlockInputSignal(0, "", lcv, idx)> * %<gain>;
  %endroll
%endfunction %% Outputs
```

Arguments for `%roll`

The lines between `%roll` and `%endroll` may be either repeated or looped. The key to understanding the `%roll` directive is in its arguments:

```
%roll sigIdx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
```

Argument	Description
`sigIdx`	Specify the index into a (signal) vector that is used in the generated code. If the signal is scalar, when analyzing that block of the *`model`*`.rtw` file, TLC determines that only a single line of code is required. In this case, it sets `sigIdx` to `0` so as to access only the first element of a vector, and no loop is constructed.
`lcv`	A control variable generally specified in the `%roll` directive as `lcv = RollThreshold`. `RollThreshold` is a global (model-

Argument	Description
	wide) threshold with the default value of 5. Therefore, whenever a block contains more than five contiguous and rollable variables, TLC collapses the lines nested between `%roll` and `%endroll` into a loop. If fewer than five contiguous rollable variables exist, `%roll` does not create a loop and instead produces individual lines of code.
`block`	This tells TLC that it is operating on block objects. TLC code for S-functions use this argument.
`"Roller"`	This, specified in `rtw/c/tlc/roller.tlc`, formats the loop. Normally you pass this as is, but other loop control constructs are possible for advanced uses (see `LibBlockInputSignal` in "TLC Function Library Reference".

`rollVars`	Tells TLC what types of items should be rolled: input signals, output signals, and/or parameters. You do not have to use all of them. In a previous line, `rollVars` is defined using `%assign`.
	
	`%assign rollVars = ["U", "Y"]`
	This list tells TLC that it is rolling through input signals (`U`) and output signals (`Y`). In cases where blocks specify an array of parameters instead of a scalar parameter, `rollvars` is specified as
	
	`%assign rollVars = ["U", "Y", "P"]`

Input Signals, Output Signals, and Parameters

Look at the lines that appear between `%roll` and `%endroll`:

```
%<LibBlockOutputSignal(0, "", lcv, idx)> = \
%<LibBlockInputSignal (0, "", lcv, idx)> * 2.0;
```

The TLC library functions `LibBlockInputSignal` and `LibBlockOutputSignal` expand to produce scalar or vector identifiers that are named and indexed. `LibBlockInputSignal`, `LibBlockOutputSignal`, and a number of related TLC functions are passed four canonical arguments:

Argument	Description
first argument — `0`	Corresponds to the input port index for a given block. The first input port has index

Argument	Description
	0. The second input port has index 1, and so on.
second argument — `" "`	An index variable reserved for advanced use. For now, specify the second argument as an empty string. In advanced applications, you may define your own variable name to be used as an index with `%roll`. In such a case, TLC declares this variable as an integer in an location in the generated code.
third argument — `lcv`	As described previously, `lcv = RollThreshold` is set in `%roll` to indicate that a loop be constructed whenever `RollThreshold` (default value of 5) is exceeded.
fourth argument — `sigIdx`	Enables TLC to handle special cases. In the event that the `RollThreshold` is *not* exceeded (for example, if the block is only connected to a scalar input signal) TLC does not roll it into a loop. Instead, TLC provides an integer value for the index variable in a corresponding line of "inline" code. Whenever the `RollThreshold` is exceeded, TLC creates a `for` loop and uses an index variable to access inputs, outputs and parameters within the loop.

For details, see "%roll" in the TLC Directives.

Debug Your TLC Code

In this section...

`tlcdebug` Tutorial Overview

Objective: Introduces the TLC debugger. You will learn how to set breakpoints and familiarize yourself with TLC debugger commands.

Folder: *`matlabroot`*`/toolbox/rtw/rtwdemos/tlctutorial/tlcdebug`

You can cause the TLC debugger to be invoked whenever the Simulink Coder build process is invoked. In this tutorial, you use it to detect a bug in a `.tlc` file for a model called `simple_log`. The bug causes the generated code output from the standalone version of the model to differ from its simulation output. The tutorial guides you through following steps:

1. **Getting Started** — Run the model and inspect output
2. **Generate and Run Code from the Model** — Compare compiled results to original output
3. **Start the Debugger and Use Its Commands** — Things you can do with the debugger
4. **Debug `timesN.tlc`** — Find out what went wrong
5. **Fix the Bug and Verify** — Easy ways to fix bugs and verify fixes

Getting Started

1. Make `tlctutorial/tlcdebug` your current folder, so that you can use the files provided.

2 In the MATLAB Command Window, create a MEX-file for the S-function:

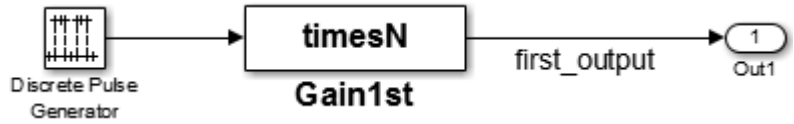
```
mex timesN.c
```

This avoids picking up the version shipped with your Simulink software.

Note: An error might occur if you have not previously run `mex -setup`.

---

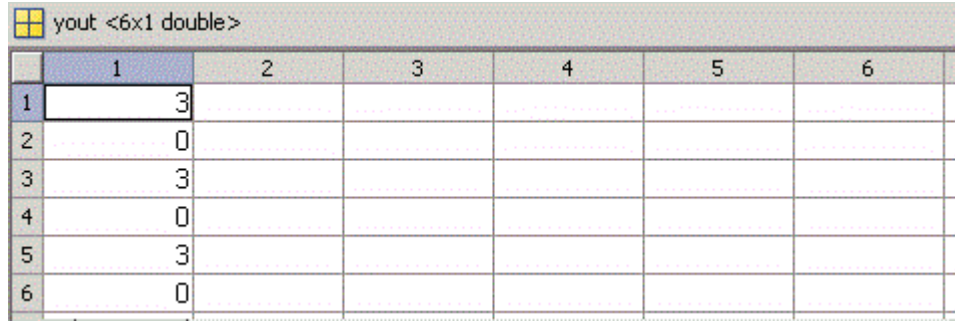
3 Open the model `simple_log`. The model looks like this.



4 In the **Data Import/Export** pane of the Configuration Parameters dialog box, under **Save to workspace**, check **Time** and **Output**. This causes model variables to be logged to the MATLAB workspace.

5 Run the model by selecting **Run** from the **Simulation** menu. Variables `tout` and `yout` appear in your MATLAB workspace.

6 Double-click `yout` in the **Workspace** pane of the MATLAB Command Window. The Variable Editor displays the 6x1 array output from `simple_log`. The display looks like this:



	1	2	3	4	5	6
1	3					
2	0					
3	3					
4	0					
5	3					
6	0					

Column 1 contains discrete pulse output for six time steps (3s and 0s), collected at port `out1`.

Next, you generate a standalone version of `simple_log`. You execute it and compare its results to the output from Simulink displayed above.

Note: For the purpose of this exercise, the TLC file provided, `timesN.tlc`, contains a bug. This version must be in the same folder as the model that uses it.

Generate and Run Code from the Model

1 In the **Code Generation** pane of the Configuration Parameters dialog box, click **Build** (or type **Ctrl+B**).

The code generator produces, compiles, and links C source code. The MATLAB Command Window shows the progress of the build, which ends with these messages:

```
### Created executable: simple_log.exe
### Successful completion of build procedure
    for model: simple_log
```

2 Run the standalone model just created by typing

```
!simple_log
```

This results in the messages

```
** starting the model **
** created simple_log.mat **
```

3 Inspect results by placing the variables in your workspace. In the **Current Folder** pane, double-click `simple_log.mat`, then double-click `rt_yout` (the standalone version of variable `yout`) in the **Workspace** pane.

Compare `rt_yout` with `yout`. Do you notice differences? Can you surmise what caused values in `rt_yout` to change?

A look at the generated C code that TLC placed in your build folder (`simple_log_grt_rtw`) helps to identify the problem.

4 Edit `simple_log.c` and look at its `MdlOutputs` function, which should appear as shown below:

```
/* Model output function */
static void simple_log_output(int_T tid)
{
  /* DiscretePulseGenerator: '<Root>/Discrete Pulse Generator' */
  simple_log_B.DiscretePulseGenerator = ((real_T)
    simple_log_DWork.clockTickCounter < 1.0) &&
    (simple_log_DWork.clockTickCounter >= 0) ? 1.0 : 0.0;
  if ((real_T)simple_log_DWork.clockTickCounter >= 2.0 - 1.0) {
    simple_log_DWork.clockTickCounter = 0;
```

```
  } else {
    simple_log_DWork.clockTickCounter = simple_log_DWork.clockTickCounter + 1;
  }

  /* S-Function Block: <Root>/Gain1st */
  /* Multiply input by 3.0 */
  simple_log_B.first_output = simple_log_B.DiscretePulseGenerator * 1;

  /* Outport: '<Root>/Out1' */
  simple_log_Y.Out1 = simple_log_B.first_output;
  UNUSED_PARAMETER(tid);
}
```

Note the line near the end:

`simple_log_B.first_output = simple_log_B.DiscretePulseGenerator * 1;`
How did a constant value get passed to the output when it was supposed to receive a variable that alternates between 3.0 and 0.0? Use the debugger to find out.

Start the Debugger and Use Its Commands

You use the TLC debugger to monitor the code generation process. As it is not invoked by default, you need to request the debugger explicitly.

1. Set up the TLC debugging environment and start to build the application:

   a. Select the **Code Generation** > **Debug** pane, and select the options **Retain .rtw file** and **Start TLC debugger when generating code**. Click **Apply**.

   b. Click **Build** on the **Code Generation** pane.

      The MATLAB Command Window describes the building process. The build stops at the `timesN.tlc` file and displays the command prompt:

      ```
      TLC-DEBUG>
      ```

2. Type `help` to list the TLC debugger commands. Here are some things you can do in the debugger.

   - View and query various entities in the TLC scope.

     ```
     TLC-DEBUG> whos CompiledModel
     TLC-DEBUG> print CompiledModel.NumSystems
     TLC-DEBUG> print TYPE(CompiledModel.NumSystems)
     ```

   - Examine the statements in your current context.

     ```
     TLC-DEBUG> list
     ```

```
TLC-DEBUG> list 10,40
```

- Move to the next line of code.

  ```
  TLC-DEBUG> next
  ```

- Step into a function.

  ```
  TLC-DEBUG> step
  ```

- Assign a constant value to a variable, such as the input signal `%<u>`.

  ```
  TLC-DEBUG> assign u = 5.0
  ```

- Set a breakpoint where you are or in some other part of the code.

  ```
  TLC-DEBUG> break timesN.tlc:10
  ```

- Execute until the next breakpoint.

  ```
  TLC-DEBUG> continue
  ```

- Clear breakpoints you have established.

  ```
  TLC-DEBUG> clear 1
  TLC-DEBUG> clear all
  ```

3 If you have tried the TLC debugger commands, execute the remaining code to finish the build process, then build `simple_log` again. The build stops at the `timesN.tlc` file and displays the command prompt:

```
TLC-DEBUG>
```

Debug `timesN.tlc`

Now look around to find out what is wrong with the code:

1 Set a breakpoint on line 20 of `timesN.tlc`.

```
TLC-DEBUG> break timesN.tlc:20
```

2 Instruct the TLC debugger to advance to your breakpoint.

```
TLC-DEBUG> continue
```

TLC processes input, reports its progress, advances to line 20 in `timesN.tlc`, displays the line, and pauses.

```
### Loading TLC function libraries
...
### Initial pass through model to cache user defined code
```

```
.
### Caching model source code
.
Breakpoint 1
00020:   %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
```

3 Use the `whos` command to see the variables in the current scope.

```
TLC-DEBUG> whos
Variables within: <BLOCK_LOCAL>
gain                            Real
rollVars                        Vector
block                           Resolved
system                          Resolved
```

4 Inspect the variables using the print command (names are case sensitive).

```
TLC-DEBUG> print gain
3.0

TLC-DEBUG> print rollVars
[U, Y]
```

5 Execute one step.

```
TLC-DEBUG> step
00021:     %<LibBlockOutputSignal(0, "", lcv, idx)> = \
```

6 Because it is a built-in function, advance via the `next` command.

```
TLC-DEBUG> next
.
00022:     %<LibBlockInputSignal(0, "", lcv, idx)> * 1;
```

This is the origin of the C statement responsible for the erroneous constant output, `simple_log_B.first_output = simple_log_B.DiscretePulseGenerator * 1;`.

7 Abandon the build by quitting the TLC debugger. Type

```
TLC-DEBUG> quit
```

An error message is displayed showing that you stopped the build by using the TLC debugger `quit` command. Close the error window.

Fix the Bug and Verify

The problem you identified is caused by evaluating a constant rather than a variable inside the TLC function `FcnEliminateUnnecessaryParams()`. This is a typical coding error and is easily repaired. Here is the code you need to fix.

```
%function Outputs(block, system) Output
```

```
   %assign gain =SFcnParamSettings.myGain
  /* %<Type> Block: %<Name> */
  %%
  /* Multiply input by %<gain> */
  %assign rollVars = ["U", "Y"]
  %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
    %<LibBlockOutputSignal(0, "", lcv, idx)> = \
    %<LibBlockInputSignal(0, "", lcv, idx)> * 1;
  %endroll

%endfunction

%% [EOF] timesN.tlc
```

1. To fix the coding error, edit `timesN.tlc`. The line

   ```
   %<LibBlockInputSignal(0, "", lcv, idx)> * 1;
   ```

   multiplies the evaluated input by 1. Change the line to

   ```
   %<LibBlockInputSignal(0, "", lcv, idx)> * %<gain>;
   ```

   Save `timesN.tlc`.

2. Build the standalone model again. Complete the build by typing `continue` at each `TLC-DEBUG>` prompt.

3. Execute the standalone model by typing

   ```
   !simple_log
   ```

   A new version of `simple_log.mat` is created containing its output.

4. Load `simple_log.mat` and compare the workspace variable `rt_yout` with `yout`, as you did before. The values in the first column should now correspond.

For more information about the TLC debugger, see “Debugging”.

TLC Code Coverage to Aid Debugging

In this section...

`tlcdebug` Execute Tutorial Overview

Objective: Learn to use TLC coverage statistics to help identify bugs in TLC code.

Folder: *`matlabroot`*`/toolbox/rtw/rtwdemos/tlctutorial/tlcdebug`

This tutorial teaches you how to determine whether your TLC code is being executed as expected. Here it uses the same model as for the previous tutorial. As you focus on understanding flow of control in processing TLC files, you don't need to compile and execute a standalone model, only to look at code. The tutorial proceeds as follows:

1. **Getting Started** — Why and how to analyze TLC coverage
2. **Open the Model and Generate Code** — Read a coverage log file

Getting Started

The **Code Generation** > **Debug** pane provides the option **Start TLC coverage when generating code**. Selecting it results in a listing that documents how many times each line in your TLC source file was executed during code generation. The listing, *`name`*`.log` (where *`name`* is the filename of the TLC file being analyzed), is placed in your build folder.

Note: A log file for every `.tlc` file invoked or included is generated in the build folder. Focus on `timesN.log`.

Open the Model and Generate Code

1. Copy the folder `tlctutorial/tlcdebug/` to your working folder and `cd` to it. *Do this even though you already have copied it*, to be sure you have the version of `timesN.tlc` that has the bug.

2. In the MATLAB Command Window, create a MEX-file for the S-function.

   ```
   mex timesN.c
   ```

   This avoids picking up the version shipped with your Simulink software.

3. Open the model `simple_log`.

4. In the **Code Generation** pane of the Configuration Parameters dialog box, check **Generate code only**.

5. In the **Code Generation** > **Debug** pane of the Configuration Parameters dialog box, select **Start TLC coverage when generating code**. (Do not select **Start TLC debugger when generating code**. Invoking the debugger is unnecessary.) Click **Apply**.

6. In the **Code Generation** pane of the Configuration Parameters dialog box, click **Generate Code**. The usual messages appear in the MATLAB Command Window, and a build folder (`simple_log_grt_rtw`) is created in your working folder.

7. Enter the build folder. Find the file `timesN.log`, and copy it to your working folder, renaming it to `timesN_ilp.log` to prevent it from being overwritten.

8. Open the log file `timesN_ilp.log` in your editor. It looks almost like `timesN.tlc`, except for a number followed by a colon at the beginning of each line. This number represents the number of times TLC executed the line in generating code. The code for `Outputs()` should look like this:

   ```
   0: %% Function: Outputs ================================================================
        0: %%
        1: %function Outputs(block, system) Output
        1:   %assign gain =SFcnParamSettings.myGain
        1:   /* %<Type> Block: %<Name> */
        0:   %%
        1:   /* Multiply input by %<gain> */
        1:   %assign rollVars = ["U", "Y"]
        1:   %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
        1:     %<LibBlockOutputSignal(0, "", lcv, idx)> = \
        1:     %<LibBlockInputSignal(0, "", lcv, idx)> * 1;
        0:   %endroll
        1:
        0: %endfunction
   ```

   Notice that comments were not executed. TLC statements were reached, which means they output to the generated C code as many times as the number prefixed to those lines.

Changing code generation options can cause a latent issue in generated source code. Systematically changing options and observing the resulting differences in TLC coverage can facilitate the process of discovering faulty code.

Wrap User Code with TLC

In this section...

`wrapper` Tutorial Overview

Objective: Learn the architecture of wrapper S-functions and how to create an inlined wrapper S-function using TLC.

Folder: *`matlabroot`*`/toolbox/rtw/rtwdemos/tlctutorial/wrapper`

Wrapper S-functions enable you to use existing C functions without fully rewriting them in the context of Simulink S-functions. Each wrapper you provide is an S-function “shell” that merely calls one or more existing, external functions. This tutorial explains and illustrates wrappers as follows:

- **Why Wrap User Code?** — Reason for building TLC wrapper functions
- **Getting Started** — Set up the wrapper exercise
- **Generate Code Without a Wrapper** — How the Simulink Coder code generator handles external functions by default
- **Generate Code Using a Wrapper** — Bypass the API overhead

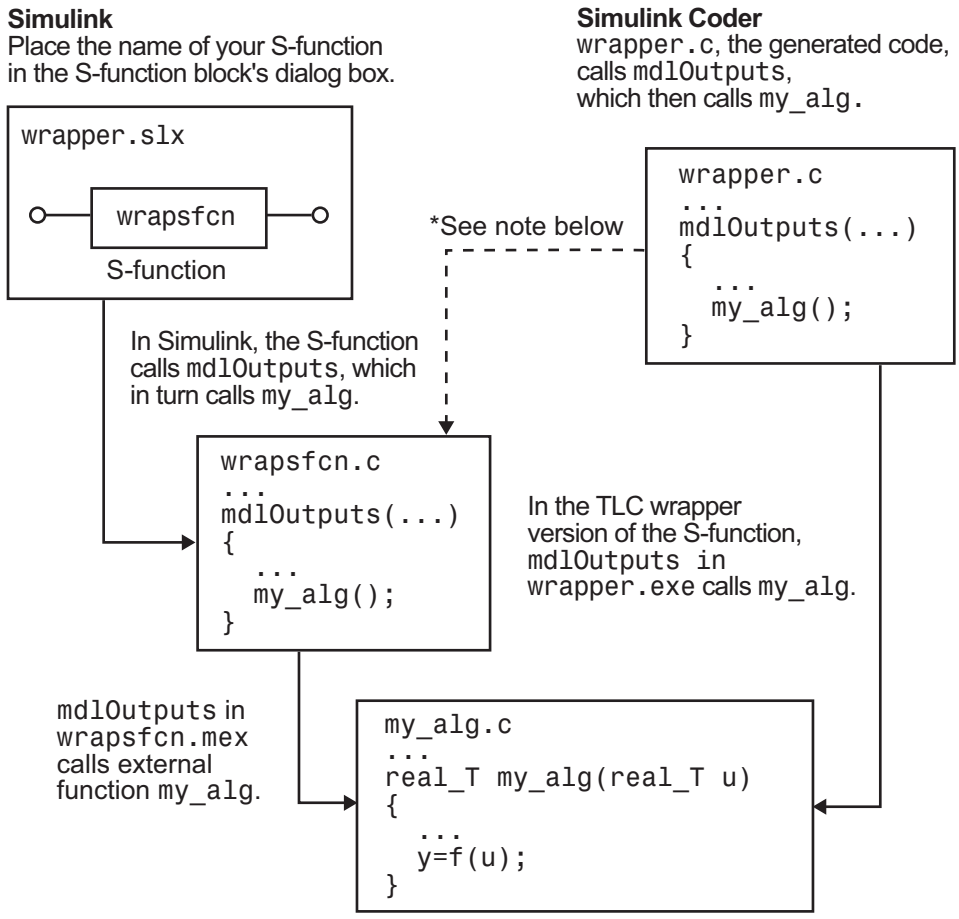
Why Wrap User Code?

Many Simulink users want to build models incorporating algorithms that they have already coded, implemented, and tested in a high-level language. Typically, such code is brought into Simulink as S-functions. To generate an external application that integrates user code, you can take several approaches:

- You can construct an S-function from user code that hooks it to the Simulink generic API. This is the simplest approach, but sacrifices efficiency for standalone applications.

- You can inline the S-function, reimplementing it as a TLC file. This improves efficiency, but takes time and effort, can introduce errors into working code, and leads to two sets of code to maintain for each algorithm, unless you use the Legacy Code Tool (see “Integrate External Code Using Legacy Code Tool”).
- You can inline the S-function via a TLC *wrapper function*. By doing so, you need to create only a small amount of TLC code, and the algorithm can remain coded in its existing form.

The next figure illustrates how S-function wrappers operate.



*The dotted line is the path taken if the S-function does not have a TLC wrapper file. If there is no TLC wrapper file, the generated code calls `mdlOutputs`.

Wrapping a function eliminates the need to recode it, requiring only a bit of extra TLC code to integrate it. Wrappers also enable object modules or libraries to be used in S-functions. This may be the only way to deploy functions for which source code is unavailable, and also allows users to distribute models to others without divulging implementation details that may be proprietary.

For example, you might have an existing object file compiled for a processor on which Simulink does not run. You can write a dummy C S-function and use a TLC wrapper that

calls the external function, despite not having its source code. You could similarly access functions in a library of algorithms optimized for the target processor. Accomplishing this requires making changes to a template makefile, or otherwise providing a means to link against the library.

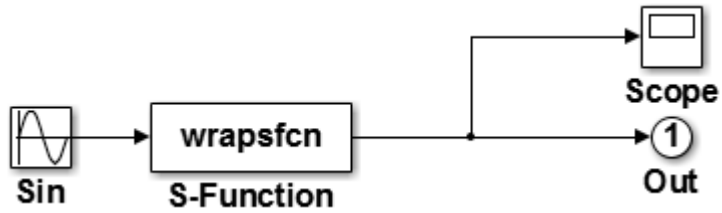
Note: Object files that lack source code and are created with Microsoft® Visual C and Microsoft Visual C++® Compiler (MSVC) work only with MSVC.

The only restriction on S-function wrappers is for the number of block inputs and outputs match number of inputs and outputs of the wrapped external function. Wrapper code may include computations, but usually these are limited to transforming values (for example, scaling or reformatting) passed to and from the wrapped external functions.

Getting Started

In the example folder, the "external function" is found in the file `my_alg.c`. You are also provided with a C S-function called `wrapsfcn.c` that integrates `my_alg.c` into Simulink. Set up the exercise as follows:

1. Make `tlctutorial/wrapper` your current folder.
2. In MATLAB, open the model `externalcode` from your working folder. The block diagram looks like this:



3. Activate the Scope block by double-clicking it.
4. Run the model (from the **Simulation** menu, or type **Ctrl+T**). You will get an error telling you that `wrapsfcn` does not exist. Can you figure out why?
5. The error occurs because a `mex` file does not exist for `wrapsfcn`. To rectify this, in the MATLAB Command Window type

```
mex wrapsfcn.c
```

Note: An error might occur if you have not previously run `mex -setup`.

6 Run the simulation again with the S-function present.

The S-Function block multiplies its input by two. Looking at the Scope block, you see a sine wave that oscillates between -2.0 and 2.0. The variable `yout` that is created in your MATLAB workspace steps through these values.

In the remainder of the exercise, you build and run a standalone version of the model, then write some TLC code that allows the code generator to build a standalone executable that calls the S-function `my_alg.c` directly.

Generate Code Without a Wrapper

Before creating a wrapper, generate code that uses the Simulink generic API. The first step is to build a standalone model.

1 Choose **Code** > **C/C++ Code** > **Build Model**.

The code generator creates the standalone program in your working folder and places the source and object files in your build folder. The file will be called `externalcode.exe` on Microsoft Windows® platforms or `externalcode` on UNIX platforms.

As it generates the program, the code generator reports its progress in the MATLAB Command Window. The final lines are:

```
### Created executable: externalcode.exe
### Successful completion of build procedure
for model: externalcode
```

2 Run the standalone program to see that it behaves the same as the Simulink version. There should not be differences.

```
!externalcode

** starting the model **
** created externalcode.mat **
```

3 Notice this line in `wrapsfcn.c`:

```
#include "my_alg.c"
```

This pulls in the external function. That function consists entirely of

```
/*
 * Copyright 1994-2002 The MathWorks, Inc.
 */

double my_alg(double u)
{
    return(u * 2.0);
}
```

Inspect the `mdlOutputs()` function of the code in `wrapsfcn.c` to see how the external function is called.

```
static void mdlOutputs(SimStruct *S, int tid)
{
    int_T             i;
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
    real_T            *y    = ssGetOutputPortRealSignal(S,0);
    int_T             width = ssGetOutputPortWidth(S,0);

    *y = my_alg(*uPtrs[0]);
}
```

Generally, functions to be wrapped are either included in the wrapper, as above, or, when object modules are being wrapped, resolved at link time.

Generate Code Using a Wrapper

To create a wrapper for the external function `my_alg.c`, you need to construct a TLC file that embodies its calling function, `wrapsfcn.c`. The TLC file must generate C code that provides:

- A function prototype for the external function that returns a double, and passes the input `double u`.
- A function call to `my_alg()` in the outputs section of the code.

To create a wrapper for `my_alg()`, do the following:

1. Open the file `change_wrapsfcn.tlc` in your editor, and add lines of code where comments indicate to create a workable wrapper.

2. Save the edited file as `wrapsfcn.tlc`. It must have the same name as the S-function block that uses it or TLC is not called to inline code.
3. In MATLAB, open the model `externalcode` from your working folder. Activate the Scope block by double-clicking it, and run the model (from the **Simulation** menu, or type **Ctrl+T**). This gives you a baseline result.
4. Inform Simulink that your code has an external reference to be resolved. To update the model's parameters, in the MATLAB Command Window, do one of the following:

   - Type

     ```
     set_param('externalcode/S-Function','SFunctionModules','my_alg')
     ```

   - In the S-Function block parameters dialog box, in the `S-function modules` field, specify `'my_alg'`.

5. Create the standalone application, by entering one of the following commands in the Command Window:

   ```
   rtwbuild('my_alg','ForceTopModelBuild',true)
   ```

   ```
   slbuild('my_alg', 'StandaloneRTWTarget','ForceTopModelBuild',true)
   ```

   These commands force the coder to rebuild the top model, which is required when you make changes associated with external or custom code.

   Alternatively, you can force regeneration of top model code by deleting code generation folders, such as `slprj` or the generated model code folder.

   For more information, see "Control Regeneration of Top Model Code".

6. Run the new standalone application and verify that it yields identical results as in the scope window.

   ```
   !externalcode
   ```

If you had problems building the application:

- Find the error messages and try to determine what files are at fault, paying attention to which step (code generation, compiling, linking) failed.
- Be sure you issued the `set_param()` command as specified above.
- Chances are that problems can be traced to your TLC file. It may be helpful to use TLC debugger to step through `wrapsfcn.tlc`.

- As a last resort, look at `wrapsfcn.tlc` in the `solutions/tlc_solution` folder, also listed below:

```
%% File    : wrapsfcn.tlc
%% Abstract:
%%      Example tlc file for S-function wrapsfcn.c
%%
%% Copyright 1994-2002 The MathWorks, Inc.
%%
%%

%implements "wrapsfcn" "C"

%% Function: BlockTypeSetup ================================
%% Abstract:
%%      Create function prototype in model.h as:
%%     "extern double my_alg(double u);"
%%
%function BlockTypeSetup(block, system) void
  %openfile buffer
  %% ASSIGNMENT: PROVIDE A LINE OF CODE AS A FUNCTION PROTOTYPE
  %% FOR "my_alg" AS DESCRIBED IN THE WRAPPER TLC ASSIGNMENT
  extern double my_alg(double u);

  %closefile buffer
  %<LibCacheFunctionPrototype(buffer)>
%endfunction %% BlockTypeSetup

%% Function: Outputs =======================================
%% Abstract:
%%      y = my_alg( u );
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign u = LibBlockInputSignal(0, "", "", 0)
  %assign y = LibBlockOutputSignal(0, "", "", 0)
  %% PROVIDE THE CALLING STATEMENT FOR "wrapfcn"
  %<y> = my_alg( %<u> );
%endfunction %% Outputs
```

  Look at the highlighted lines. Did you declare `my_alg()` as `extern double`? Did you call `my_alg()` with the expected input and output? Fix mistakes and rebuild the model.

4

Code Generation Architecture

Build Process

In this section...

Build Process Overview

TLC compiles files written in the target language. The target language is an interpreted language and the compiler operates on source files every time it executes. You can make changes to a target file and watch the effects of your change the next time you build a model. You do not need to recompile TLC binary or other large binary to see the changes.

Because the target language is an interpreted language, some statements might not be compiled or executed (and hence not checked by the compiler). For example:

```
%if 1
   Hello
%else
   %<Invalid_function_call()>
%endif
```

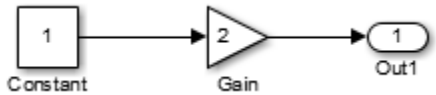
In this example, the `Invalid_function_call` statement will not be executed. This example emphasizes that you should test TLC code with test cases that execute every line.

Create and Use Target Language File

This example creates a target language file that generates specific text from a Simulink Coder model. It shows the sequence of steps that you should follow in creating and using your own target language files.

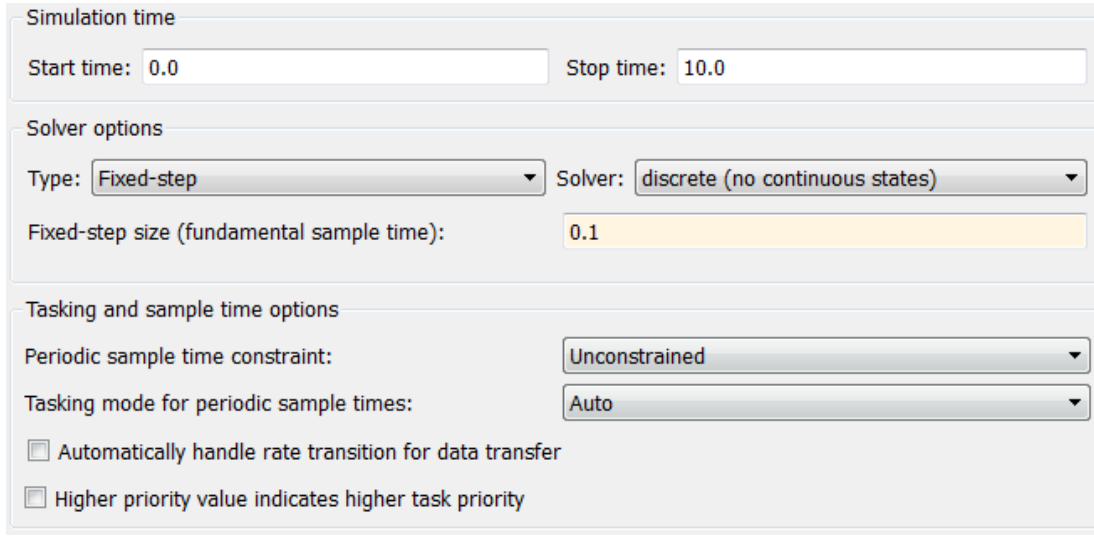
Process

To begin, create the Simulink model shown in the next figure.



1. Save the new model in a working folder as `basic`.
2. Display the Configuration Parameters dialog box.
3. Select the **Solver** pane.
4. In the **Solver** pane:
   a. Select `Fixed-step` in the **Type** field.
   b. Select `discrete (no continuous states)` in the **Solver** field.
   c. Specify `0.1` in the **Fixed-step size** field. (Otherwise, the code generator posts a warning and supplies a value when you generate code.)

   The **Solver** pane should now look like this:



5. Click **Apply**.
6. Select the **Code Generation** > **Debug** pane.
7. Select **Retain .rtw file**, then click **Apply**. This step lets you inspect the contents of the *`model`*`.rtw` file after the build finishes.
8. Select the **Code Generation** pane.
9. Select **Generate code only**, then click **Apply**.
10. Click **Generate code**.

The build process generates code in the `basic_grt_rtw` folder. You can see the progress in the MATLAB Command Window. When code generation is complete, following message is displayed:

```
### Successful completion of code generation for model: basic
```

The `slbuild` Command

Typically, you invoke `slbuild` directly from the Simulink Coder build procedure by clicking the **Build** (or **Generate code**) button on the **Code Generation** pane of the Configuration Parameters dialog box. However, some circumstances may require you to execute `slbuild` directly from the MATLAB prompt.

To generate a *`model`*`.rtw` file from the MATLAB prompt, type:

```
slbuild('model')
```

You can specify other options to `slbuild` that build or rebuild model reference simulation targets or a stand-alone executable. For more information, type:

```
help slbuild
```

at the MATLAB prompt or see `slbuild` in the Simulink documentation.

Viewing the `basic.rtw` file

A *`model`*`.rtw` file contains a hierarchy of labeled records and fields. Each record is delimited by brackets, and contains subordinate records and/or fields. The labels state the purpose of each record and field. The records and fields in the *`model`*`.rtw` file created for a model describe various details of the model and the Configuration Parameter settings that specify its context.

Open the file `./basic_grt_rtw/basic.rtw`, in MATLAB or a text editor. The following example is a short extract of the file. The extract is intended only to show the general appearance of a *`model`*`.rtw` file. Your file, `basic.rtw`, will contain many more records and fields, sometimes with different field values than appear in the extract.

```
CompiledModel {
  Name     "basic"
  OrigName    "basic"
  Version    "8.3 (R2012b) 20-Jul-2012"
  SimulinkVersion   "8.0"
  ModelVersion    "1.4"
  GeneratedOn    "Fri Aug 03 19:33:05 2012"
  HasSimStructVars   0
  HasCodeVariants   0
  PreserveExternInFcnDecls 1
  ExprFolding    1
```

```
  TargetStyle    "StandAloneTarget"
  NumDataStoresGlobalDSM  0
  RightClickBuild   0
  ModelReferenceTargetType "NONE"
...
  ConfigSet {
    AutosarCompliant     0
    BitfieldContainerType   "uint_T"
    BlockReduction     1
    BooleanDataType     0
    BooleansAsBitfields     0
    BufferReusableBoundary  1
    BufferReuse      1
    CPPClassGenCompliant    0
    CodeExecutionProfiling  0
    ...
  }
  Solver    FixedStepDiscrete
  SolverType    FixedStep
  StartTime    0.0
  StopTime    10.0
...
  FixedStepOpts {
    SolverMode      SingleTasking
    FixedStep      0.1
...
  }
  RTWGenSettings {
    BuildDirSuffix     "_grt_rtw"
    CodeFormat      "RealTime"
    DisableBuildDirOverride "no"
    DivideStackByRate     "0"
...
  }
  DataLoggingOpts {
    SaveFormat      0
    MaxRows      1000
    Decimation      1
    TimeSaveName     "tout"
    OutputSaveName     "yout"
...
  }
  NumModelInputs   0
  NumModelOutputs   1
...
  AllSampleTimesInherited yes
...
  BlockParamChecksum   ["2593893983U", "3970349032U", "3137491486U", "2062188880U"]
  ModelChecksum    ["1708110901U", "2264152010U", "1036968531U", "586837897U"]
...
}
```

Create the Target File

Note: The following exercise is provided to give a conceptual overview of how the `.rtw` file is used in the Simulink Coder build process. In general, the code generator does not support manually invoking TLC with a `.rtw` file created from an earlier Simulink Coder build. Additionally, the contents of the `.rtw` file are undocumented and subject to change. The `basic.tlc` file is used to show how information provided in a `.rtw` file can be accessed by the TLC files and executed as part of the Simulink Coder build process.

Next, create a `basic.tlc` file to act as a target file for this model. Instead of generating code, simply display some information about the model using this file. The concept is the same as used in code generation.

Create a file called `basic.tlc` in the folder containing `basic`. This file should contain the following lines:

```
%with CompiledModel

My model is called %<Name>.
It was generated on %<GeneratedOn>.
It has %<NumModelOutputs> output(s) and %<NumContStates> continuous state(s).

%endwith
```

Note: In the Simulink Coder build process, the `.tlc` file specified on the command line when TLC is invoked (for example, `grt.tlc`) is referred to as the System Target File (STF). It can be selected via the **System target file** browser option in the **Code Generation** pane of the Configuration Parameters dialog box.

In this example, you generate the `.rtw` file as part of the Simulink Coder build process and then manually run TLC using the file `basic.tlc` as an example STF. `basic.tlc` illustrates (in a limited capacity) how `.rtw` file information is used to generate an example output. To do this, enter at the MATLAB prompt:

```
slbuild('basic')
tlc -r basic_grt_rtw/basic.rtw basic.tlc -v
```

The first line generates the `.rtw` file in the build folder `'basic_grt_rtw'`. This step is actually unnecessary because the file has already been generated in the previous step; however, it will be useful if the model is changed and the operation has to be repeated.

The second line runs TLC on the file `basic.tlc`. The `-r` option tells TLC that it should use the file `basic.rtw` as the `.rtw` file. Note that a space must separate `-r` and the input filename. The `-v` option tells TLC to be verbose in reporting its activity.

The output of this pair of commands is (date will differ):

```
My model is called basic.
It was generated on Wed Jun 22 20:51:11 2005.
It has 1 output(s) and 0 continuous state(s).
```

You can also try changing the model (for instance, by using `rand(2,2)` as the value for the constant block) and then repeating the process to see how the output of TLC changes.

As you continue through this chapter, you will learn more about creating target files.

Configure TLC

In this section...

Set Command-Line Arguments

You can enter TLC command-line arguments from the MATLAB command line using the `set_param` command, the model parameter `TLCOptions`, and the TLC option `-a`. For example, to enter the TLC command-line string `-amyConfigVariable=1`, use the following MATLAB command:

```
set_param(modelName,'TLCOptions','-amyConfigVariable=1');
```

Using `-amyConfigVariable=1` is equivalent to coding the following in your target file:

```
%assign myConfigVariable = 1
```

Alternatively, you can configure the TLC code generation process by using the `-a` option on the TLC command line. That is, you must give the TLC command interactively.

You can repeatedly use the `-a` option.

For an example of how this process works, consider the following TLC code fragment:

```
%if !EXISTS(myConfigVariable)
  %assign myConfigVariable = 0
%endif
  %if (myConfigVariable == 1)
      code fragment 1
  %else
      code fragment 2
  %endif
```

If you specify `-amyConfigVariable=1` in the command line, `code fragment 1` is generated; otherwise `code fragment 2` is generated. The `if` block starting with

```
%if !EXISTS(myConfigVariable)
```

serves to set the default value of `myConfigVariable` to 0, so that TLC does not generate an error if you forget to add `-amyConfigVariable` to the command line.

If you use the `-a` option to input a string variable, the variable must be enclosed in double quotation marks:

```
-aMyStringVariable="hello"
```

However, if the string contains white space, enclose the string within apostrophes and double quotation marks:

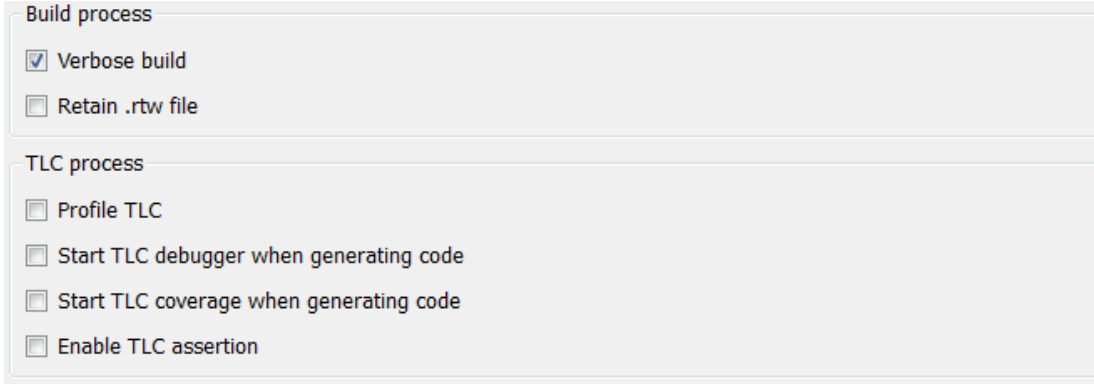
```
-aMyStringVariable="'hello world'"
```

You must also do this if apostrophes exist within the string, whether or not white space is included, and the apostrophes must be escaped (doubled):

```
-aMyStringVariable="'can''t'"
```

Configure for TLC Debugging

To configure TLC for debugging via the Configuration Parameters dialog, select the **Debug** pane under **Code Generation**. This provides the following TLC process options for configuring the build process:



The **Start TLC debugger when generating code** check box lets you activate the TLC debugger . This is covered in more detail in “Debugging”.

Code Generation Concepts

In this section...

Overview

TLC interprets a target language, which is a general programming language, and you can use it as such. It is important, however, to remember that TLC was designed for one purpose: to convert a *`model`*`.rtw` file to generated code. Thus, the target language provides many features that are particularly useful for this task but does not provide some of the features that other languages like C and C++ provide.

Before you start modifying or creating target files, you might find some of the following general programming examples useful to familiarize yourself with the basic constructs used within TLC.

Output Streams

The typical “Hello World” example is rather simple in the target language. Type the following in a file named `hello.tlc`:

```
%selectfile STDOUT
Hello, World
```

To run this TLC program, type

```
tlc hello.tlc
```

at the MATLAB prompt.

This simple script illustrates some important concepts underlying the purpose (and hence the design) of TLC. Since the primary purpose of TLC is to generate code, it is output (or stream) oriented. It makes it easy to handle buffers of text and output them easily. In the above script, the `%selectfile` directive tells TLC to send any following

text that it generates or does not recognize to the standard output device. Syntax that TLC recognizes begins with the `%` character. Because `Hello, World` is not recognized, it is sent directly to the output. You could just as easily change the output destination to be a file. The `STDOUT` stream does not have to be opened, but must be selected to write to the Command Window.

```
%openfile foo = "foo.txt"
%openfile bar = "bar.txt"
%selectfile foo
This line is in foo.
%selectfile STDOUT
Line has been output to foo.
%selectfile bar
This line is in bar.
%selectfile NULL_FILE
This line will not show up anywhere.
%selectfile STDOUT
About to close bar.
%closefile bar
%closefile foo
```

Note that you can switch between buffers to display status messages. The semantics of the three directives `%openfile`, `%selectfile`, and `%closefile` are given in "Target Language Compiler Directives".

Variable Types

The absence of explicit type declarations for variables is another feature of TLC. See "Directives and Built-In Functions" for more information on the implicit data types of variables.

Records

One of the constructs most relevant to generating code from the *`model`*`.rtw` file is a record. A *record* is very similar to a structure in C or a record in Pascal. The syntax of a record declaration is

```
%createrecord recVar { ...
   field1  value1  ...
   field2  value2 ...
   ...
  fieldN  valueN ...
```

```
}
```

where `recVar` is the name of the record being declared, `fieldi` is a string, and `valuei` is the corresponding TLC value.

Records can have nested records, or subrecords, within them. The *`model`*`.rtw` file is essentially one large record, named `CompiledModel`, containing levels of subrecords.

Unlike MATLAB, TLC requires that you explicitly load function definitions not located in the same target file. In MATLAB, the line `A = myfunc(B)` causes MATLAB to automatically search for and load a MATLAB file or MEX-file named `myfunc`. TLC, on the other hand, requires that you specifically include the file that defines the function using the `%addincludepath` directive.

TLC provides a `%with` directive that facilitates using records. See "Directives and Built-In Functions" for a detailed description TLC directives.

Note The format and structure of the *`model`*`.rtw` file are subject to change from one release of the Simulink Coder product to another.

A record read in from a file is changeable, like any other record that you might declare in a program. In fact, the Simulink Coder record `CompiledModel` is modified many times during code generation. `CompiledModel` is the global record in the *`model`*`.rtw` file. It contains variables used for code generation, such as `NumNonvirtSubsystems`, `NumBlocks`, etc. It is also appended during code generation with many new variables, options, and subrecords.

Functions such as `LibGetFormattedBlockPath` are provided in TLC libraries located in *`matlabroot`*`/rtw/c/tlc/lib/*.tlc`. For a complete list of available functions, refer to "TLC Function Library Reference".

Assign Values to Fields of Records

To assign a value to a field of a record, you must use a *qualified variable expression*. A qualified variable expression references a variable in one of the following forms:

- An identifier
- A qualified variable followed by "`.`" followed by an identifier, such as

  ```
  var[2].b
  ```

- A qualified variable followed by a bracketed expression such as

  `var[expr]`

Record Aliases

In TLC it is possible to create what is called an *alias* to a record. Aliases are similar to pointers to structures in C. You can create multiple aliases to a single record. Modifications to the aliased record are visible to every place that holds an alias.

The following code fragment illustrates the use of aliases:

```
%createrecord foo { field 1 }
%createrecord a { }
%createrecord b { }
%createrecord c { }

%addtorecord a foo foo
%addtorecord b foo foo
%addtorecord c foo { field 1 }

%% notice we are not changing field through a or b.
%assign foo.field  = 2

ISALIAS(a.foo) = %<ISALIAS(a.foo)>
ISALIAS(b.foo) = %<ISALIAS(b.foo)>
ISALIAS(c.foo) = %<ISALIAS(c.foo)>

a.foo.field = %<a.foo.field>
b.foo.field = %<b.foo.field>
c.foo.field = %<c.foo.field>
%% note that c.foo.field is unchanged
```

Saving this script as `record_alias.tlc` and invoking it with

```
tlc -v record_alias.tlc
```

produces the output

```
ISALIAS(a.foo) = 1
ISALIAS(b.foo) = 1
ISALIAS(c.foo) = 0

a.foo.field = 2
```

```
b.foo.field = 2
c.foo.field = 1
```

When inside a function, it is possible to create an alias to a locally created record that is within the function. If the alias is returned from the function, it remains valid even after exiting the function, as in the following example:

```
%function func(value) Output
  %createrecord foo { field value }
  %createrecord a { foo foo }
ISALIAS(a.foo) = %<ISALIAS(a.foo)>
   %return a.foo
 %endfunction

 %assign x = func(2)
 ISALIAS(x) = %<ISALIAS(x)>
 x = %<x>
 x.field = %<x.field>
```

Saving this script as `alias_func.tlc` and invoking it with

```
tlc -v alias_func.tlc
```

produces the output

```
ISALIAS(a.foo) = 1
ISALIAS(x) = 1
x = { field 2 }
x.field = 2
```

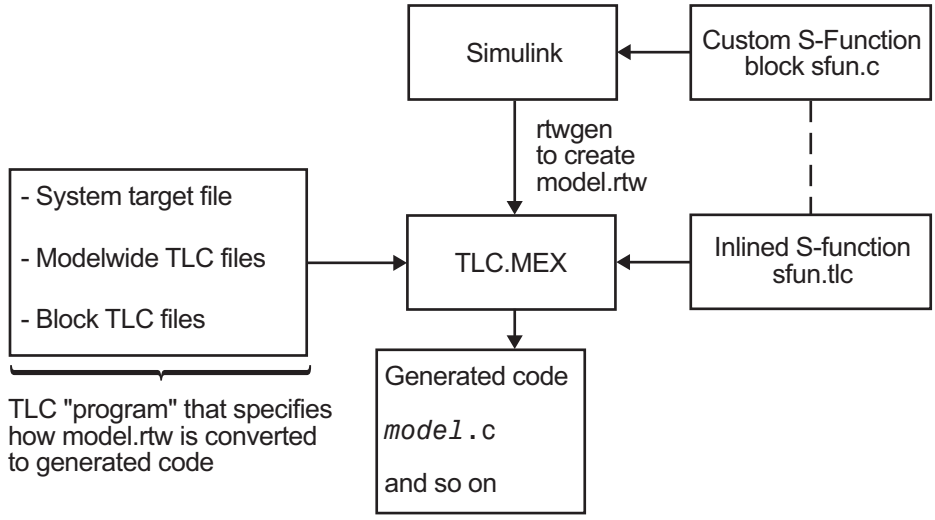
As long as there is some reference to a record through an alias, that record is not deleted. This allows records to be used as return values from functions.

TLC Files

In this section...

TLC Program

TLC works with the Simulink software to generate code as shown in the following figure.



Just as a C program is a collection of ASCII files connected with `#include` statements and object files linked into one binary, a *TLC program* is a collection of ASCII files, also called *scripts*. Because TLC is an interpreted language, there are no object files. The single target file that calls (with the `%include` directive) other target files used for the program is called the *entry point*.

Available Target Files

Target files are the set of files that are interpreted by TLC to transform the intermediate Simulink Coder code (*model*`.rtw`) produced by Simulink into target-specific code.

Target files provide you with the flexibility to customize the code generated by the compiler to suit your specific needs. For example, if you use the available system target files, you produce generic C or C++ code from your Simulink model. This executable code is not platform specific.

Note: You should not customize TLC files even though the capability exists to do so. Such TLC customizations might not be applied during the code generation process and can lead to unpredictable results. Only customize TLC files you create.

The parameters used in the target files are read from the *model*`.rtw` file and looked up using block scoping rules. You can define additional parameters within the target files, using the `%assign` statement.

Target files are written using target language directives. “Directives and Built-In Functions” provides complete descriptions of the block scope rules and the target language directives.

“model.rtw file” describes this file, which is useful for creating and/or modifying target files.

Model-Wide Target Files and System Target Files

Model-wide target files are used on a model-wide basis and provide basic information to TLC, which transforms the *model*`.rtw` file into target-specific code.

The system target file is the entry point for TLC. It is analogous to the `main()` routine of a C program. System target files oversee the entire code generation process. For example, the system target file `grt.tlc` sets up some variables for `codegenentry.tlc`, which is the entry point into the Simulink Coder target files. For a complete list of available system target files, see “Available Targets” in the Simulink Coder documentation.

Summary of Target File Usage

In the context of code generation, there are two types of target files, system target files, and block target files:

- System target files

  System target files determine the overall framework of code generation. They determine when blocks are executed, how data is logged, and so on.

- Inline an S-function

  Inlining an S-function means writing a target file that tells TLC how to generate code for that S-Function block. The compiler can automatically generate code for noninlined C MEX S-functions. However, if you inline a C MEX S-function, the compiler can generate more efficient code. Noninlined C MEX S-functions execute using the S-function application program interface (API) and can be inefficient. You can inline a MATLAB file or Fortran S-function; TLC can generate code for the S-function in both these cases.

- Customize the code generated for all models

  You might want to instrument the generated code for profiling, or make other changes to overall code generation for all models. To accomplish such changes, you must modify some of the system target files.

System Target Files

The entire code generation process starts with the single system target file that you specify in the **Code Generation** pane of the Configuration Parameters dialog box. Normally, you click the **Browse** button to activate the system target file browser for this purpose. A close examination of a system target file reveals how code generation occurs. This is a listing of the noncomment lines in `grt.tlc`, the target file to generate code for a generic real-time executable:

```
%selectfile NULL_FILE
%assign TargetType = "RT"
%assign Language   = "C"
%assign MatFileLogging = 1
%include "codegenentry.tlc"
```

The three variables, `Language`, `TargetType`, and `MatFileLogging`, are global TLC variables used by other functions. Code generation is then initiated with the call to `codegenentry.tlc`, the main entry point for Simulink Coder code generation.

If you want to make changes to modify overall code generation, you must change the system target file. After the initial setup, instead of calling `codegenentry.tlc`, you

must call your own TLC files. The following code shows an example system target file called `mygrt.tlc`.

```
%% Set up variables, etc.
%% Load my library functions
%% Note that mylib.tlc should %include funclib.tlc at the
%% beginning.
%include "mylib.tlc"
%include "commonsetup.tlc"

%% Next, you can include TLC files that you need for
%% preprocessing information about the model and to fill in
%% hooks. The following is an example of including a single
%% TLC file that contains custom hooks.
%include "myhooks.tlc"

%% Finally, call the code generator.
%include "commonentry.tlc"
```

Generated code is placed in a model or subsystem function. The relevant generated function names and their execution order are described in “About Model Execution” and “Entry-Point Functions and Scheduling”. During code generation, functions from each of the block target files are executed and the generated code is placed in model or subsystem functions.

Data Handling with TLC

In this section...

Matrix Parameters

MATLAB, Simulink, and Simulink Coder software use column-major ordering for array storage (1-D, 2-D, ...), so that the next element of an array in memory is accessed by incrementing the first index of the array. For example, these element pairs are stored sequentially in memory: `A(i)` and `A(i+1)`, `B(i,j)` and `B(i+1,j)`, `C(i,j,k)` and `C(i+1,j,k)`. For more information on the internal representation of MATLAB data, see "MATLAB Data" in the MATLAB External Interfaces document.

Simulink Coder Matrix Parameters

Simulink and Simulink Coder internal data storage formatting differs from MATLAB internal data storage formatting only in the storage of complex number arrays. In MATLAB, the real and imaginary parts are stored in separate arrays, while in the Simulink and Simulink Coder products they are stored in an "interleaved" format, where the numbers in memory alternate real, imaginary, real, imaginary, and so forth. This convention allows efficient implementations of small signals on Simulink lines and for Mux blocks and other "virtual" signal manipulation blocks (i.e., they don't actively copy their inputs, merely the references to them).

The compiled model file, *`model`*`.rtw`, represents matrices as strings in MATLAB syntax, with no implied storage format. This is so you can copy the string out of an `.rtw` file and paste it into an `.m` file and have it recognized by MATLAB.

TLC declares Simulink block matrix parameters as scalar or 1-D array variables

```
real_T scalar;
real_T mat[ nRows * nCols ];
```

where `real_T` can be an arbitrary data type supported by Simulink, and will match the variable type given in the model file.

For example, the 3-by-3 matrix in the Look-Up Table (2-D) block

```
1   2   3
```

```
4   5   6
7   8   9
```

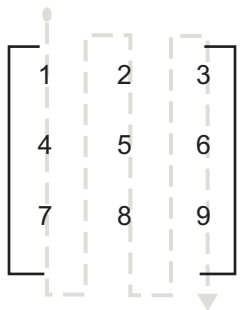
is stored in *model*.rtw as

```
Parameter {
   Name    "OutputValues"
   Value    Matrix(3,3)
[[1.0, 2.0, 3.0]; [4.0, 5.0, 6.0]; [7.0, 8.0, 9.0];]
   String    "t"
   StringType    "Variable"
   ASTNode {
     IsNonTerminal       0
    Op       SL_NOT_INLINED
     ModelParameterIdx       3
   }
}
```

and results in this definition in *model*.h

```
typedef struct Parameters_tag {
  real_T s1_Look_Up_Table_2_D_Table[9];
              /* Variable:s1_Look_Up_Table_2_D_Table
                * External Mode Tunable:yes
                   * Referenced by block:
                * <S1>/Look-Up Table (2-D
                         */

  [ ... other parameter definitions ... ]

} Parameters;
```

The *model*.h file declares the actual storage for the matrix parameter and you can see that the format is column-major. That is, read down the columns, then across the rows.



```
Parameters model_P = {
  /* 3 x 3 matrix s1_Look_Up_Table_2_D_Table */
  { 1.0, 4.0, 7.0, 2.0, 5.0, 8.0, 3.0, 6.0, 9.0 },
  [ ... other parameter declarations ...]
};
```

TLC accesses matrix parameters via `LibBlockMatrixParameter` and `LibBlockMatrixParameterAddr`, where

`LibBlockMatrixParameter(OutputValues, "", "", 0, "", "", 1)` returns `"model_P.s1_Look_Up_Table_2_D_Table[nRows]"` (automatically optimized from `"[0+nRows*1]"`) and

`LibBlockMatrixParameterAddr(OutputValues, "", "", 0, "", "", 1)` returns `"&model_P.s1_Look_Up_Table_2_D_Table[nRows]"` for both inlined and noninlined block TLC code.

Matrix parameters are like other TLC parameters in that only those parameters explicitly accessed by a TLC library function during code generation are placed in the parameters structure. So, following the example, `s1_Look_Up_Table_2_D_Table` is not declared unless it is explicitly accessed by `LibBlockParameter` or `LibBlockParameterAddr`.

5

model.rtw File and Authoring S-Functions and Data Objects

Introduction to the *model*.rtw File

The code generation software creates a *model*.rtw file from your Simulink model. A *model*.rtw file is an intermediate representation of a model generated by the build process for use by the Target Language Compiler. It describes blocks, inputs, outputs, parameters, states, storage, and other model components and properties from the corresponding model file.

The generated *model*.rtw file is input to the Target Language Compiler. If you select **Retain .rtw file** on the **Code Generation** > **Debug** pane of the Configuration Parameters dialog box, after building a model, you can view the *model*.rtw file that was generated.

A *model*.rtw file is implemented is an ASCII file of parameter-value pairs stored in a hierarchy of records. A parameter name/parameter value pair is specified as

```
ParameterName  value
```

where `ParameterName` (also called an *identifier*) is the name of the TLC identifier and `value` is a string, scalar, vector, or matrix. For example, in the parameter name/parameter value pair

```
    NumDataOutputPorts 1
```

`NumDataOutputPorts` is the identifier and 1 is its value.

A *record* is specified as

```
RecordName {
    .
    .
    .
}
```

A record contains parameter name/parameter value pairs and/or subrecords. For example, this record contains one parameter name/parameter value pair:

```
DataStores {
    NumDataStores     0
}
```

Note The structure of the *model*.rtw file is very likely to change between releases, which is a compelling reason to limit your access to *model*.rtw to the TLC library functions documented in "TLC Function Library Reference". For additional information, see "Library Functions that Access model.rtw" on page 5-14.

Scopes in the *model*.rtw File

Each record creates a new *scope*. The *model*.rtw file uses curly braces { and } to open and close records (or scopes). Using scopes, you can access values within the *model*.rtw file.

The scope in this example begins with CompiledModel. Use periods (.) to access values within particular scopes. The format of *model*.rtw is

```
CompiledModel {
  Name   "modelname"           -- Example of a parameter-value
  ...                            pair (record field).
  System {                     -- There is one system for each
                                 nonvirtual subsystem.
    Block {                    -- Block records for each
      Type     "S-Function"      nonvirtual block in the
                                 system.
      Name     "<S3>/S-Function"
      ...
      Parameter {
       Name "P1"
       Value Matrix(1,2) [[1, 2];]
    }
    ...
    Block {
    }
  }
  ...
  System {                     -- The last system is for the
                                 root of your model.
}
```

For example, to access Name within CompiledModel, you would use

```
CompiledModel.Name
```

Multiple records of the same name form a list where the index of the first record starts at 0. To access the above S-function block record, you would use

```
CompiledModel.System[0].Block[0]
```

To access the name field of this block, you would use

```
CompiledModel.System[0].Block[0].Name
```

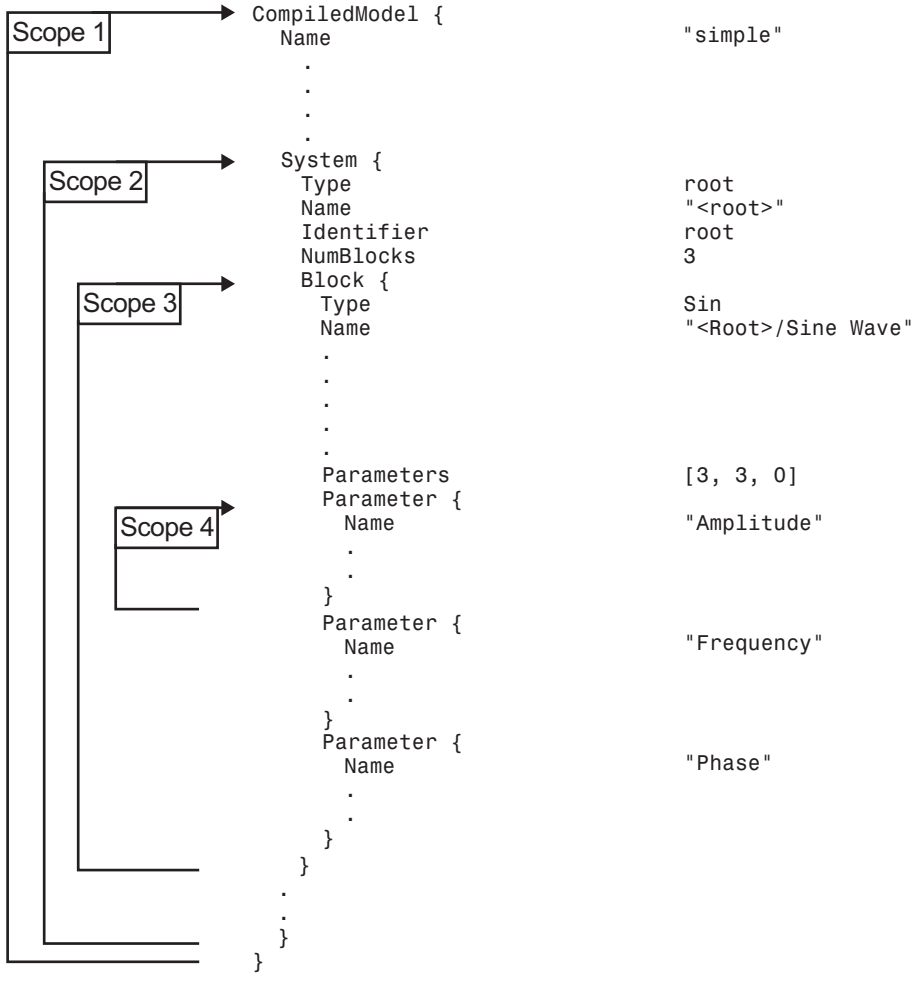
To simplify this process, you can use the `%with` directive, which changes the current scope. For example:

```
%with CompiledModel.System[0].Block[0]
%assign blockName = Name
%endwith
```

`blockName` will have the value `"<S3>/S-Function"`.

When inlining S-function blocks, your S-function block record is scoped as though the above `%with` directive was done. In an inlined `.tlc` file, you should access fields without a fully qualified path.

The following code shows a more detailed scoping example where the `Block` record has several parameter-value pairs (`Type`, `Name`, `Identifier`, and so on), and three subrecords, each called `Parameter`. `Block` is a subrecord of `System`, which is a subrecord of `CompiledModel`. Note that the parameter names in this file changes from release to release.


Scope 1
Scope 2
Scope 3
Scope 4
CompiledModel {
Name "simple"
System {
Type root
Name "<root>"
Identifier root
NumBlocks 3
Block {
Type Sin
Name "<Root>/Sine Wave"
Parameters [3, 3, 0]
Parameter {
Name "Amplitude"
}
Parameter {
Name "Frequency"
}
Parameter {
Name "Phase"
}
}
}

Data Object Information in *model*.rtw

In this section...

Data Object Overview

During the build process, the code generator writes information about Simulink signal and parameter data objects to the *model*.rtw file. An `Object` record with `CoderInfo` property information is written for each parameter or signal that meets certain conditions. These conditions are described in "Object Records for Parameters" on page 5-7 and "Object Records for Signals" on page 5-8.

The `Object` records contain the information corresponding to the associated data object. To access `Object` records, you must write Target Language Compiler code (see "Access Data Object Information via TLC" on page 5-10).

Note: The `Object` record examples in this section are generated from the example model `rtwdemo_advsc`, with model button **ExportedGlobal Storage Class** double-clicked and model option **Retain .rtw file** selected. (Do not use the example model buttons to build the model, as they modify model options, including **Retain .rtw file**.)

Object Records for Parameters

An `Object` record with `CoderInfo` property information is included in the `ModelParameters` section of the *model*.rtw file for each parameter that meets the following conditions:

- The parameter resolves to a `Simulink.Parameter` data object (or to a parameter data object that comes from a class derived from the `Simulink.Parameter` class).
- The parameter symbol is preserved in the generated code. The symbol is preserved when **Inline parameters** is on and `CoderInfo.StorageClass` is not set to `"Auto"` or `"SimulinkGlobal"`.

The following example shows part of an `Object` record for a parameter. A real record contains more fields than appear in the example.

```
ModelParameters {
  NumParameters     10
  ...
  Parameter {
    Identifier              "LOWER"
    LogicalSrc              P2
    WorkspaceVarName        "LOWER"
    Tunable                 yes
    StorageClass            "ExportedGlobal"
    Value                   [-10.0]
    ReferencedBy            Matrix(1,4)
[[1, -1, 1, 0];]
    GraphicalRef            Matrix(1,2)
[[0, 12];]
    OwnerSysIdx             [1, -1]
    HasObject               1
    Object {
      Package                 Simulink
      Class                   Parameter
      ObjectProperties {
        Value                   -10.0
        CoderInfo {
          Object {
            Package                 Simulink
            Class                   ParamCoderInfo
            ObjectProperties {
              StorageClass            "ExportedGlobal"
              Alias                   ""
              CustomStorageClass      "Default"
              CustomAttributes {
                Object {
                  Package                 SimulinkCSC
                  Class                   AttribClass_Simulink_Default
                  ObjectProperties {
                  }
                }
              }
            }
          }
        }
      }
  ...
}
```

Object Records for Signals

An `Object` record with `CoderInfo` property information is included in the `BlockOutputs` section of the *`model`*`.rtw` file for each signal that meets the following conditions:

- The signal resolves to a `Simulink.Signal` data object (or to a signal data object that comes from a class derived from the `Simulink.Signal` class).
- The signal symbol is preserved in the generated code. The symbol is preserved when `CoderInfo.StorageClass` is not set to `"Auto"` or `"SimulinkGlobal"`.

If the signal is configured to be an unstructured global variable in the generated code, its validity and uniqueness are enforced and its symbol is preserved.

The following example shows part of an `Object` record for a signal. A real record contains more fields than appear in the example.

```
BlockOutputs {
  ...
  NumExternalBlockOutputs 1
  ...
  ExternalBlockOutput {
    Identifier              "output"
    SysCsIdx                [0, 0]
    LogicalSrc              B7
    StorageClass            "ExportedGlobal"
    DrivesRootOutport       yes
    GrSrc                   [0, 10, 0, -1]
    SigSrc                  [0, 0, 0, 1]
    SigLabel                "output"
    HasObject               1
    Object {
      Package                 Simulink
      Class                   Signal
      ObjectProperties {
        CoderInfo {
          Object {
            Package                 Simulink
            Class                   SignalCoderInfo
            ObjectProperties {
              StorageClass            "ExportedGlobal"
              Alias                   ""
              CustomStorageClass      "Default"
              CustomAttributes {
                Object {
                  Package                 SimulinkCSC
                  Class                   AttribClass_Simulink_Default
                  ObjectProperties {
                  }
                }
              }
            }
          }
        }
      }
  ...
}
```

Access Data Object Information via TLC

This section provides sample code to illustrate how to access data object information from the *model*.rtw file using TLC code.

Access Parameter Object Records

The following code fragment iterates over `Parameter` structures in the `ModelParameters` section of the *model*.rtw file and extracts information from parameter `Object` records encountered.

```
%with CompiledModel.ModelParameters
  %foreach modelParamIdx = NumParameters
    %assign thisModelParam = Parameter[modelParamIdx]
    %assign paramName = thisModelParam.Identifier
    %if EXISTS("thisModelParam.Object.ObjectProperties")
      %with thisModelParam.Object.ObjectProperties
        %assign valueInObject = Value
        %with CoderInfo.Object.ObjectProperties
          %assign storageClassInObject  = StorageClass
        %endwith
        %% ********************************
        %% Access user-defined properties here
        %% ********************************
        %if EXISTS("MY_PROPERTY_NAME")
          %assign userDefinedPropertyName = MY_PROPERTY_NAME
        %endif
        %% ********************************
      %endwith
    %endif
  %endforeach
%endwith
```

Access Signal Object Records

The following code fragment iterates over `ExternalBlockOutput` structures in the `BlockOutputs` section of the *model*.rtw file and extracts information from signal `Object` records encountered.

```
%with CompiledModel.BlockOutputs
  %foreach blockOutputIdx = NumExternalBlockOutputs
    %assign thisBlockOutput = ExternalBlockOutput[blockOutputIdx]
    %assign signalName = thisBlockOutput.Identifier
    %if EXISTS("thisBlockOutput.Object.ObjectProperties")
      %with thisBlockOutput.Object.ObjectProperties
        %with CoderInfo.Object.ObjectProperties
          %assign storageClassInObject  = StorageClass
        %endwith \
        %% ********************************\
        %% Access user-defined properties here\
        %% ********************************
```

```
        %if EXISTS("MY_PROPERTY_NAME")
          %assign userDefinedPropertyName = MY_PROPERTY_NAME
        %endif
        %% ********************************
      %endwith
    %endif
  %endforeach
%endwith
```

Data References in the *model*.rtw File

In this section...

Data Reference Overview

Some records in a *model*.rtw file, such as those corresponding to parameters and constant block I/O, can have extremely large data value vectors embedded in them. Such a vector can cause significant memory overhead during code generation because the values must be maintained as text in memory during this process.

To avoid such overhead, by default the Simulink software does not write out the entire data value vector into *model*.rtw. Instead, it writes a key called a *data reference* that can be used during code generation to access the data directly from Simulink. If the data is not mutated during code generation, it is efficiently streamed to disk when the actual code containing the data values is written out.

A data reference has the format SLData(*index*), where *index* is a numeric value that tells Simulink which data is being referenced. TLC directives such as GENERATE_FORMATTED_VALUE store data references in unexpanded format in memory. When the generated code is written out to disk, the data values expand to the actual values.

Control the Data Reference Threshold

By default, Simulink writes a data reference to *model*.rtw in place of a data vector whose length is 10 or more. To change the maximum length of a vector that can appear literally in the file, use:

```
set_param(0, 'RTWDataReferencesMinSize', maxlen)
```

Simulink replaces a vector as long or longer than maxlen with a data reference when it creates *model*.rtw. Specify maxlen as an integer or as inf. Specifying inf disables

data references. The complete value set of every vector, however long, then appears literally in *model*.rtw and occupies text memory during code generation.

Setting an explicit *maxlen* affects only the current MATLAB session. To set the value across sessions, include a set_param command in your startup.m file, or automate execution of the command when MATLAB launches.

Expand Data References

You can explicitly expand a data reference by using the GENERATE_FORMATTED_VALUE built-in function with the optional third expand argument. Commands such as FEVAL may cause a data reference to be expanded to the full form.

Avoid Data Reference Expansion

Either turning off data references completely or expanding select parameters in TLC can cause significant text memory overhead during the code generation process. During most common code generation tasks, it is unnecessary to have the expanded data vector in memory and pay the price of the additional overhead. Avoid expanded data vectors unless no alternative exists.

Restart Code Generation

A *model*.rtw file that contains data references cannot be used in isolation to restart a custom code generation process. The data references within it become stale once the code generation process is completed. Attempting to start a code generation process using only this file may result in unpredictable behavior and memory segmentation faults.

Library Functions that Access *model*.rtw

In this section...

Library Functions Overview

There are several library functions that provide access to block inputs, outputs, parameters, sample times, and other information. It is recommended that you use these library functions to access many of the parameter name/parameter values pairs in the block record, as opposed to accessing the parameter name/parameter value pairs directly from your block TLC code.

The library functions simplify block TLC code and provide support for loop rolling, data types, and complex data. The functions also provide a layer to protect against changes that can occur to the contents of the *model*.rtw file.

Caution Against Directly Accessing Record Fields

When functions in the block target file are called, they are passed the block and system records for this instance as arguments. The first argument, `block`, is in scope, which means that variable names inside this instance's block record are accessible by name. For example:

```
%assign fast = SFcnParamSetting.Fast
```

Block target files could generate code for a given block by directly using the fields in the Block record for the block. This process is *not* recommended, for two reasons:

- The contents of the *model*.rtw file can change from release to release. This can cause block TLC files that access the *model*.rtw file directly to stop working.
- TLC library functions are provided that substantially reduce the amount of TLC code for implementing a block while handling the various configurations (widths, data types, etc.) a block might have. These library functions are provided by the system target files to provide access to inputs, outputs, parameters, and so on. Using these functions in a block TLC script makes it flexible enough to generate code for multiple

instances or configurations of the block, as well as across releases. Exceptions to this do occur, however, such as when you want to directly access a field in the block's record. This happens with parameter settings, as discussed in "TLC Code to Access the Parameter Settings" on page 5-15.

Exception to Using the Library Functions

An exception to using these functions is when you access parameter settings for a block. Parameter settings can be written out using the `mdlRTW` function of a C MEX S-function. They can contain data in the form of strings, scalar values, vectors, and matrices. They can be used to pass fixed values and information that is used to alter the generated code for a block or directly as values in the resulting code of a block.

mdlRTW Function in C MEX S-Function Code

```
static void mdlRTW(SimStruct *S)
{
    if (!ssWriteRTWParamSettings( S, 1, SSWRITE_VALUE_QSTR, "Operator", "AND"))
    {
        ssSetErrorStatus(S,"Error writing parameter data to .rtw file");
        return;
    }
}
```

Resulting Block Record in *model*.rtw File

```
Block {
      Type    "S-Function"
      Name    "<Root>/S-Function"

      ...

      SFcnParamSettings {
        Operator    "AND"
      }
    }
```

TLC Code to Access the Parameter Settings

```
%function Outputs(block, system) Output
  %%
  %% Select Operator
  %switch(SFcnParamSettings.Operator)
    %case "AND"
      %assign LogicOp        = "&"
      %break
    ...
  %endswitch
%endfunction
```

For more details on using parameter settings, see “Inlining S-Functions”.

Directives and Built-In Functions

You control how code is generated from models largely through writing or modifying scripts that apply TLC directives and built-in functions. Use the following sections as your primary reference to the syntax and format of target language constructs, as well as the MATLAB `tlc` command itself.

Target Language Compiler Directives

In this section...

Syntax

A target language file consists of a series of statements of either form

```
[text | %<expression>]*
```

```
%keyword [argument1, argument2, ...]
```

Statements of the first type cause literal text to be passed to the output stream unmodified, and expressions enclosed in `%< >` are evaluated before being written to output (stripped of `%< >`).

For statements of the second type, `%keyword` represents one of the Target Language Compiler's directives, and `[argument1, argument2, ...]` represents expressions that define required parameters. For example, the statement

```
%assign sysNumber = sysIdx + 1
```

uses the `%assign` directive to define or change the value of the `sysNumber` parameter.

A target language directive must be the first nonblank character on a line and begins with the `%` character. Lines beginning with `%%` are TLC comments, and are *not* passed to the output stream. Lines beginning with `/*` are C comments, and *are* passed to the output stream.

Directives

The rest of this section shows the complete set of Target Language Compiler directives, and describes each directive in detail.

`%% text`

Single-line comment where `text` is the comment.

`/% text%/`

Single (or multiline) comment where `text` is the comment.

`%matlab`

Calls a MATLAB function that does not return a result. For example, `%matlab disp(2.718)`.

`%<expr>`

Target language expressions that are evaluated. For example, if you have a TLC variable that was created via `%assign varName = "foo"`, then `%<varName>` would expand to `foo`. Expressions can also be function calls, as in `%<FcnName(param1,param2)>`. On directive lines, TLC expressions need not be placed within the `%<>` syntax. Doing so will cause a double evaluation. For example, `%if %<x> == 3` is processed by creating a hidden variable for the evaluated value of the variable x. The `%if` statement then evaluates this hidden variable and compares it against 3. The efficient way to do this

operation is to write `%if x == 3`. In MATLAB notation, this would equate to writing if `eval('x') == 3` as opposed to if `x = 3`. The exception to this is during an `%assign` for format control, as in

```
%assign str = "value is: %<var>"
```

Note: Nested evaluation expressions (e.g., `%<foo(%<expr>)>`) are not supported.

There is not a speed penalty for evaluations inside strings, such as

```
%assign x = "%<expr>"
```

Avoid evaluations outside strings, such as the following example.

```
%assign x = %<expr>
```

`%if` *`expr`*
`%elseif` *`expr`*
`%else`
`%endif`

Conditional inclusion, where the constant expression *`expr`* must evaluate to an integer. For example, the following code checks whether a parameter, `k`, has the numeric value 0.0 by executing a TLC library function to check for equality.

```
%if ISEQUAL(k, 0.0)
  <text and directives to be processed if k is 0.0>
%endif
```

In this and other directives, you do not have to expand variables or expressions using the `%<expr>` notation unless `expr` appears within a string. For example,

`%if ISEQUAL(idx, "my_idx%<i>")`, where `idx` and `i` are both strings.

As in other languages, logical evaluations do short-circuit (are halted as soon as the result is known).

`%switch` *`expr`*
  `%case` *`expr`*
  `%break`
  `%default`
  `%break`

%endswitch

The %switch directive is very similar to the C language switch statement. The expression expr should be of a type that can be compared for equality using the == operator. If the %break is not included after a %case statement, then it will fall through to the next statement.

%with
%endwith

%with recordName is a scoping operator. Use it to bring the named record into the current scope, to remain until the matching %endwith is encountered (%with directives can be nested as desired).

Note that on the left side of %assign statements contained within a %with / %endwith block, references to fields of records must be fully qualified (see "Assign Values to Fields of Records" on page 4-12), as in the following example.

```
%with CompiledModel
  %assign oldName = name
  %assign CompiledModel.name = "newname"
  %endwith
```

%setcommandswitch *string*

Changes the value of a command-line switch as specified by the argument string. Only the following switches are supported:

```
v, m, p, O, d, r, I, a
```

The following example sets the verbosity level to 1.

```
%setcommandswitch "-v1"
```

See also "Command-Line Arguments" on page 6-64.

%assert *expr*

Tests a value of a Boolean expression. If the expression evaluates to false, TLC issues an error message, a stack trace and exit; otherwise, the execution continues normally. To enable the evaluation of assertions outside the Simulink Coder environment, use the command-line option -da. When building from within the code generator, this flag is ignored, as it is superseded by the **Enable TLC assertion** check box on the **TLC**

process section of the **Code Generation** > **Debug** pane. To control assertion handling from the MATLAB Command Window, use

```
set_param(model, 'TLCAssertion', 'on|off')
```

to set this flag on or off. Default is Off. To see the current setting, use

```
get_param(model, 'TLCAssertion')
```

%error
%warning
%trace
%exit

Flow control directives:

`%error` *`tokens`*

The *`tokens`* are expanded and displayed.

`%warning` *`tokens`*

The *`tokens`* are expanded and displayed.

`%trace` *`tokens`*

The *`tokens`* are expanded and displayed only when the verbose output command-line option `-v` or `-v1` is specified.

`%exit` *`tokens`*

The *`tokens`* are expanded, displayed, and TLC exits.

When reporting errors, use the following command if the error is produced by an incorrect configuration that the user needs to fix in the model.

```
%exit Error Message
```

If you are adding assert code (that is, code that should never be reached), use

```
%setcommandswitch "-v1" %% force TLC stack trace
%exit Assert message
```

%assign

Creates identifiers (variables). The general form is

```
%assign [::]variable = expression
```

The `::` specifies that the variable being created is a global variable; otherwise, it is a local variable in the current scope (i.e., a local variable in the function).

If you need to format the variable, say, within a string based upon other TLC variables, then you should perform a double evaluation, as in

```
%assign nameInfo = "The name of this is %<Name>"
```

or alternately

```
%assign nameInfo = "The name of this is " + Name
```

To assign a value to a field of a record, you must use a qualified variable expression. See "Assign Values to Fields of Records" on page 4-12.

%createrecord

Creates records in memory. This command accepts a list of one or more record specifications (e.g., `{ foo 27 }`). Each record specification contains a list of zero or more name-value pairs (e.g., `foo 27`) that become the members of the record being created. The values themselves can be record specifications, as the following illustrates.

```
%createrecord NEW_RECORD { foo 1 ; SUB_RECORD {foo 2} }
%assign x = NEW_RECORD.foo                  /* x = 1 */
%assign y = NEW_RECORD.SUB_RECORD.foo       /* y = 2 */
```

If more than one record specification follows a given record name, the set of record specifications constitutes an array of records.

```
%createrecord RECORD_ARRAY { foo 1 } ...
              { foo 2 } ...
              { bar 3 }
%assign x = RECORD_ARRAY[1].foo             /* x = 2 */
%assign y = RECORD_ARRAY[2].bar             /* y = 3 */
```

Note that you can create and index arrays of subrecords by specifying `%createrecord` with identically named subrecords, as follows:

```
%createrecord RECORD_ARRAY { SUB_RECORD { foo 1 } ...
                             SUB_RECORD { foo 2 } ...
                             SUB_RECORD { foo 3 } }
%assign x = RECORD_ARRAY.SUB_RECORD[1].foo  /* x = 2 */
%assign y = RECORD_ARRAY.SUB_RECORD[2].foo  /* y = 3 */
```

If the scope resolution operator (`::`) is the first token after the `%createrecord` token, the record is created in the global scope.

Note: You should not create a record array by using `%createrecord` within a loop.

%addtorecord

Adds fields to an existing record. The new fields can be name-value pairs or aliases to already existing records.

```
%addtorecord OLD_RECORD foo 1
```

If the new field being added is a record, then `%addtorecord` makes an alias to that record instead of a deep copy. To make a deep copy, use `%copyrecord`.

```
%createrecord NEW_RECORD { foo 1 }
%addtorecord OLD_RECORD NEW_RECORD_ALIAS NEW_RECORD
```

%mergerecord

Adds (or merges) one or more records into another. The first record will contain the results of the merge of the first record plus the contents of the other records specified by the command. The contents of the second (and subsequent) records are deep copied into the first (i.e., they are not references).

```
%mergerecord OLD_RECORD NEW_RECORD
```

If duplicate fields exist in the records being merged, the original record's fields are not overwritten.

%copyrecord

Makes a deep copy of an existing record. It creates a new record in a similar fashion to `%createrecord` except the components of the record are deep copied from the existing record. Aliases are replaced by copies.

```
%copyrecord NEW_RECORD OLD_RECORD
```

%realformat

Specifies how to format real variables. To format in exponential notation with 16 digits of precision, use

```
%realformat "EXPONENTIAL"
```

To format without loss of precision and minimal number of characters, use

```
%realformat "CONCISE"
```

When inlining S-functions, the format is set to `concise`. You can switch to `exponential`, but should switch it back to `concise` when done.

%language

This must appear before the first `GENERATE` or `GENERATE_TYPE` function call. This specifies the name of the language as a string, which is being generated as in `%language "C"`. Generally, this is added to your system target file.

The only valid value is `C` which enables support for `C` and `C++` code generation as specified by the configuration parameter `TargetLang` (see "Language" for more information).

%implements

Placed within the `.tlc` file for a specific record type, when mapped via `%generatefile`. The syntax is `%implements "Type" "Language"`. When inlining an S-function in C or C++, this should be the first noncomment line in the file, as in

```
%implements "s_function_name" "C"
```

The next noncomment lines will be `%function` directives specifying the functionality of the S-function.

See the `%language` and `GENERATE` function descriptions for further information.

%generatefile

Provides a mapping between a record `Type` and functions contained in a file. Each record can have functions of the same name but different contents mapped to it (i.e., polymorphism). Generally, this is used to map a `Block` record `Type` to the `.tlc` file that implements the functionality of the block, as in

```
%generatefile "Sin" "sin_wave.tlc"
```

%filescope

Limits the scope of variables to the file in which they are defined. A `%filescope` directive anywhere in a file declares that variables in the file are visible only within that file. Note that this limitation also applies to files inserted, via the `%include` directive, into the file containing the `%filescope` directive.

You should not use the `%filescope` directive within functions or `GENERATE` functions.

`%filescope` is useful in conserving memory. Variables whose scope is limited by `%filescope` go out of scope when execution of the file containing them completes. This frees memory allocated to such variables. By contrast, global variables persist in memory throughout execution of the program.

`%include`

Use `%include "file.tlc"`to insert the specified target file at the current point.

The `%include` directives behave as if they were in a global context. For example,

```
%addincludepath "./sub1"
%addincludepath "./sub2"
```

in a `.tlc` file enables either subfolder to be referenced implicitly:

```
%include "file_in_sub1.tlc"
%include "file_in_sub2.tlc"
```

Use forward slashes for folder names, as they work on both UNIX and PC systems. However, if you do use back slashes in PC folder names, be sure to escape them, e.g., `"C:\\mytlc"`. Alternatively, you can express a PC folder name as a literal using the L format specifier, as in `L"C:\mytlc"`.

`%addincludepath`

Use `%addincludepath "folder"` to add additional paths to be searched. Multiple `%addincludepath` directives can appear. The compiler evaluates multiple `%addincludepath` directives from the *bottom up*.

Using `%addincludepath` directives establishes a global context. For example,

```
%addincludepath "./sub1"
%addincludepath "./sub2"
```

in a `.tlc` file enables either subfolder to be referenced implicitly:

```
%include "file_in_sub1.tlc"
%include "file_in_sub2.tlc"
```

Use forward slashes for folder names, as they work on both UNIX and PC systems. However, if you do use back slashes in PC folder names, be sure to escape them, e.g.,

`"C:\\mytlc"`. Alternatively, you can express a PC folder name as a literal using the L format specifier, as in `L"C:\mytlc"`.

`%roll`
`%endroll`

Multiple inclusion plus intrinsic loop rolling based upon a specified threshold. This directive can be used by most Simulink blocks that have the concept of an overall block width that is usually the width of the signal passing through the block.

This example of the `%roll` directive is for a gain operation, `y=u*k`:

```
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign rollVars = ["U", "Y", "P"]
  %roll sigIdx = RollRegions, lcv = RollThreshold, block,...
        "Roller", rollVars
    %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
    %assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
    %assign k = LibBlockParameter(Gain, "", lcv, sigIdx)
      %<y> = %<u> * %<k>;
  %endroll
%endfunction
```

The `%roll` directive is similar to `%foreach`, except that it iterates the identifier (`sigIdx` in this example) over roll regions. Roll regions are computed by looking at the input signals and generating regions where the inputs are contiguous. For blocks, the variable `RollRegions` is automatically computed and placed in the `Block` record. An example of a roll regions vector is `[0:19, 20:39]`, where there are two contiguous ranges of signals passing through the block. The first is `0:19` and the second is `20:39`. Each roll region is either placed in a loop body (e.g., the C language `for` statement) or inlined, depending upon whether or not the length of the region is less than the roll threshold.

Each time through the `%roll` loop, `sigIdx` is an integer for the start of the current roll region or an offset relative to the overall block width when the current roll region is less than the roll threshold. The TLC global variable `RollThreshold` is the general model-wide value used to decide when to place a given roll region in a loop. When the decision is made to place a given region in a loop, the loop control variable is a valid identifier (e.g., `"i"`); otherwise it is `""`.

The `block` parameter is the current block that is being rolled. The `"Roller"` parameter specifies the name for internal `GENERATE_TYPE` calls made by `%roll`. The default `%roll` handler is `"Roller"`, which is responsible for setting up the default block loop rolling structures (e.g., a C `for` loop).

The `rollVars` (roll variables) are passed to `"Roller"` functions to create roll structures. The defined loop variables relative to a block are

`"U"`

The inputs to the block. It assumes you use `LibBlockInputSignal(portIdx, "", lcv, sigIdx)` to access each input, where `portIdx` starts at 0 for the first input port.

`"u`*`i`*`"`

Similar to `"U"`, except only for specific input, *`i`*. The `"u"` must be lowercase or it will be interpreted as `"U"` above.

`"Y"`

The outputs of the block. It assumes you use `LibBlockOutputSignal(portIdx, "", lcv, sigIdx)` to access each output, where `portIdx` starts at 0 for the first output port.

`"y`*`i`*`"`

Similar to `"Y"`, except only for specific output, *`i`*. The `"y"` must be lowercase or it will be interpreted as `"Y"` above.

`"P"`

The parameters of the block. It assumes you use `LibBlockParameter(name, "", lcv, sigIdx)` to access them.

`"<param>/`*`name`*`"`

Similar to `"P"`, except specific for a specific *`name`*.

`rwork`

The `RWork` vectors of the block. It assumes you use `LibBlockRWork(name, "", lcv, sigIdx)` to access them.

`"<rwork>/`*`name`*`"`

Similar to `RWork`, except for a specific *`name`*.

`dwork`

The `DWork` vectors of the block. It assumes you use `LibBlockDWork(name, "", lcv, sigIdx)` to access them.

`"<dwork>/`*`name`*`"`

Similar to `DWork`, except for a specific *`name`*.

`iwork`

The `IWork` vectors of the block. It assumes you use `LibBlockIWork(name, "", lcv, sigIdx)` to access them.

`"<iwork>/`*`name`*`"`

Similar to `IWork`, except for a specific *`name`*.

`pwork`

The `PWork` vectors of the block. It assumes you use `LibBlockPWork(name, "", lcv, sigIdx)` to access them.

`"<pwork>/`*`name`*`"`

Similar to `PWork`, except for a specific *`name`*.

`"Mode"`

The mode vector. It assumes you use `LibBlockMode("",lcv,sigIdx)` to access it.

`"PZC"`

Previous zero-crossing state. It assumes you use `LibPrevZCState("",lcv, sigIdx)` to access it.

To *roll* your own vector based upon the block's roll regions, you need to walk a pointer to your vector. Assuming your vector is pointed to by the first `PWork`, called `name`,

```
datatype *buf = (datatype*)%<LibBlockPWork(name,"","",0)
%roll sigIdx = RollRegions, lcv = RollThreshold, block, ...
      "Roller", rollVars
  *buf++ = whatever;
%endroll
```

Note: In the above example, `sigIdx` and `lcv` are local to the body of the loop.

%breakpoint

Sets a breakpoint for the TLC debugger. See "%breakpoint Directive" on page 7-6.

%function
%return
%endfunction

A function that returns a value is defined as

```
%function name(optional-arguments)
    %return value
  %endfunction
```

A void function does not produce output and is not required to return a value. It is defined as

```
%function name(optional-arguments) void
  %endfunction
```

A function that produces outputs to the current stream and is not required to return a value is defined as

```
%function name(optional-arguments) Output
  %endfunction
```

For block target files, you can add to your inlined `.tlc` file the following functions that are called by the model-wide target files during code generation.

`%function` **`BlockInstanceSetup`**`(block, system) void`

Called for each instance of the block within the model.

`%function` **`BlockTypeSetup`**`(block, system) void`

Called once for each block type that exists in the model.

`%function` **`Enable`**`(block, system) Output`

Use this if the block is placed within an enabled subsystem and has to take specific actions when the subsystem is enabled. Place within a subsystem enable routine.

`%function` **`Disable`**`(block, system) Output`

Use this if the block is placed within a disabled subsystem and has to take specific actions when the subsystem is disabled. Place within a subsystem disable routine.

`%function` **`Start`**`(block, system) Output`

Include this function if your block has startup initialization code that needs to be placed within `MdlStart`.

`%function` **`InitializeConditions`**`(block, system) Output`

Use this function if your block has state that needs to be initialized at the start of execution and when an enabled subsystem resets states. Place in `MdlStart` and/or subsystem initialization routines.

`%function` **`Outputs`**`(block, system) Output`

The primary function of your block. Place in `MdlOutputs`.

`%function` **`Update`**`(block, system) Output`

Use this function if your block has actions to be performed once per simulation loop, such as updating discrete states. Place in `MdlUpdate`.

`%function` **`Derivatives`**`(block,system) Output`

Use this function if your block has derivatives.

`%function` **`ZeroCrossings`**`(block,system) Output`

Use this function if your block does zero-crossing detection and has actions to be performed in `MdlZeroCrossings`.

`%function` **`Terminate`**`(block, system) Output`

Use this function if your block has actions that need to be in `MdlTerminate`.

`%foreach`
`%endforeach`

Multiple inclusion that iterates from `0` to the `upperLimit-1` constant integer expression. Each time through the loop, the `loopIdentifier`, (e.g., `x`) is assigned the current iteration value.

```
%foreach loopIdentifier = upperLimit
   %break -- use this to exit the loop
   %continue -- use this to skip the following code and
                continue to the next iteration
%endforeach
```

Note: The `upperLimit` expression is cast to a TLC integer value. The `loopIdentifier` is local to the loop body.

`%for`

Multiple inclusion directive with syntax

```
%for ident1 = const-exp1, const-exp2, ident2 = const-exp3
  %body
    %break
    %continue
  %endbody
%endfor
```

The first portion of the `%for` directive is identical to the `%foreach` statement. The `%break` and `%continue` directives act the same as they do in the `%foreach` directive. `const-exp2` is a Boolean expression that indicates whether the loop should be rolled (see `%roll` above).

If `const-exp2` evaluates to `TRUE`, `ident2` is assigned the value of `const-exp3`. Otherwise, `ident2` is assigned an empty string.

Note: `ident1` and `ident2` above are local to the loop body.

`%openfile`
`%selectfile`
`%closefile`

These are used to manage the files that are created. The syntax is

```
%openfile streamId="filename.ext" mode  {open for writing}
%selectfile streamId                    {select an open file}
%closefile streamId                     {close an open file}
```

Note that the `"filename.ext"` is optional. If a filename is not specified, a variable (string buffer) named `streamId` is created containing the output. The mode argument is optional. If specified, it can be `"a"` for appending or `"w"` for writing.

Note that the special string `streamIdNULL_FILE` specifies no output. The special string `streamIdSTDOUT` specifies output to the terminal.

To create a buffer of text, use

```
%openfile buffer
text to be placed in the 'buffer' variable.
%closefile buffer
```

Now `buffer` contains the expanded text specified between the `%openfile` and `%closefile` directives.

`%generate`

`%generate blk fn` is equivalent to `GENERATE(`*`blk`*`,`*`fn`*`)`.

`%generate blk fn type` is equivalent to `GENERATE(`*`blk`*`,`*`fn`*`,`*`type`*`)`.

See “GENERATE and GENERATE_TYPE Functions” on page 6-33.

`%undef`

`%undef` *`var`* removes the variable *`var`* from scope. If *`var`* is a field in a record, `%undef` removes that field from the record. If *`var`* is a record array, `%undef` removes the entire record array.

Comments

You can place comments anywhere within a target file. To include comments, use the `/%...%/` or `%%` directives. For example:

```
/%
    Abstract:    Return the field with [width], if field is wide
%/
```

or

```
%endfunction %% Outputs function
```

Use the `%%` construct for line-based comments; characters from `%%` to the end of the line become a comment.

Nondirective lines, that is, lines that do not have `%` as their first nonblank character, are copied into the output buffer verbatim. For example,

```
/* Initialize sysNumber */
int sysNumber = 3;
```

copies both lines to the output buffer.

To include comments on lines that do not begin with the `%` character, you can use the `/%...%/` or `%%` comment directives. In these cases, the comments are not copied to the output buffer.

Note If a nondirective line appears within a function, it is not copied to the output buffer unless the function is an output function or you specifically select an output file using the `%selectfile` directive. For more information about functions, see "Target Language Functions" on page 6-60.

Line Continuation

You can use the C language \ character or the MATLAB sequence ... to continue a line. If a directive is too long to fit conveniently on one line, this allows you to split the directive onto multiple lines. For example:

```
%roll sigIdx = RollRegions, lcv = RollThreshold, block,\
      "Roller", rollVars
```

or

```
%roll sigIdx = RollRegions, lcv = RollThreshold, block,...
      "Roller", rollVars
```

Note Use \ to suppress line feeds to the output and the ellipsis (...) to indicate line continuation. Note that \ and the ellipsis (...) cannot be used inside strings.

Target Language Value Types

This table shows the types of values you can use within the context of expressions in your target language files. Expressions in the Target Language Compiler must use these types.

Value Type String	Example	Description
`"Boolean"`	`1==1`	Result of a comparison or other Boolean operator. The result will be `TLC_TRUE` or `TLC_FALSE`.
`"Complex"`	`3.0+5.0i`	64-bit double-precision complex number (`double` on the target machine)
`"Complex32"`	`3.0F+5.0Fi`	32-bit single-precision complex number (`float` on the target machine)
`"File"`	`%openfile x`	String buffer opened with `%openfile`
`"File"`	`%openfile x = "out.c"`	File opened with `%openfile`
`"Function"`	`%function foo...`	User-defined function and `TLC_FALSE` otherwise
`"Gaussian"`	`3+5i`	32-bit integer imaginary number (`int` on the target machine)

Value Type String	Example	Description
"Identifier"	abc	Identifier values can appear only within the *model*.rtw file and cannot appear in expressions (within the context of an expression, identifiers are interpreted as values). To compare against an identifier value, use a string; the identifier will be converted to a string.
"Matrix"	Matrix (3,2) [[1, 2]; [3 , 4]; [5, 6]]	Matrices are simply lists of vectors. The individual elements of the matrix do not need to be the same type, and can be any supported type except vectors or matrices. The Matrix (3,2) text in the example is optional.
"Number"	15	Integer number (int on the target machine)
"Range"	[1:5]	Range of integers between 1 and 5, inclusive
”Real"	3.14159	Floating-point number (double on the target machine), including exponential notation
"Real32"	3.14159F	32-bit single-precision floating-point number (float on the target machine)
"Scope"	Block { ... }	Block record
"Special"	FILE_EXISTS	Special built-in function, such as FILE_EXISTS
"String"	"Hello, World"	ASCII character strings. In all contexts, two strings in a row are concatenated to form the final value, as in "Hello, " "World", which is combined to form "Hello, World". These strings include the ANSI C standard escape sequences such as \n, \r, \t, etc. Use of line continuation characters (i.e., \ and ...) inside strings is illegal.
”Subsystem"	<sub1>	Subsystem identifier. Within the context of an expansion, be careful to escape the delimiters on a subsystem identifier, as in %<x == <sub\>>.
"Unsigned"	15U	32-bit unsigned integer (unsigned int on the target machine)

Value Type String	Example	Description
`"Unsigned Gaussian"`	`3U+5Ui`	32-bit complex unsigned integer (`unsigned int` on the target machine)
`"Vector"`	`[1, 2] or BR Vector(2) [1, 2]`	Vectors are lists of values. The individual elements of a vector do not need to be the same type, and can be of any supported type except vectors or matrices.

Target Language Expressions

You can include an expression of the form `%<expression>` in a target file, the Target Language Compiler replaces `%<expression>` with a calculated replacement value based upon the type of the variables within the `%<>` operator. Integer constant expressions are folded and replaced with the resultant value; string constants are concatenated (e.g., two strings in a row `"a" "b"`, are replaced with `"ab"`).

```
%<expression>                /* Evaluates the expression.
  * Operators include most standard C
  * operations on scalars. Array indexing
  * is required for certain parameters that
  * are block-scoped within the .rtw file.*/
```

Within the context of an expression, each identifier must evaluate to an identifier or function argument currently in scope. You can use the `%< >` directive on a line to perform text substitution. To include the > character within a replacement, you must escape it with a \ character, as in

```
%<x \> 1 ? "ABC" : "123">
```

Operators that need the > character to be escaped are the following:

Operator	Description	Example
>	greater than	`y = %<x \> 2>;`
>=	greater than or equal to	`y = %<x \>= 3>;`
>>	right shift	`y = %<x \>\> 4>;`

The table Target Language Expressions lists the operators that are allowed in expressions. In this table, expressions are listed in order from highest to lowest precedence. The horizontal lines distinguish the order of operations.

As in C expressions, conditional operators are short-circuited. If the expression includes a function call with effects, the effects are noticed as if the entire expression was not fully evaluated. For example:

```
%if EXISTS(foo) && foo == 3
```

If the first term of the expression evaluates to a Boolean false (i.e., `foo` does not exist), the second term (`foo == 3`) is not evaluated.

In the following table, note that *`numeric`* is one of the following:

- `Boolean`
- `Number`
- `Unsigned`
- `Real`
- `Real32`
- `Complex`
- `Complex32`
- `Gaussian`
- `UnsignedGaussian`

Also, note that *`integral`* is one of the following:

- `Number`
- `Unsigned`
- `Boolean`

See "TLC Data Promotions" on page 6-25 for information on the promotions that result when the Target Language Compiler operates on mixed types of expressions.

Target Language Expressions

Expression	Definition
`constant`	Constant parameter value. The value can be a vector or matrix.
`variable-name`	Valid in-scope variable name, including the local function scope, if any, and the global scope

Expression	Definition
`::variable-name`	Used within a function to indicate that the function scope is ignored when the variable is looked up. This accesses the global scope.
`expr[expr]`	Index into an array parameter. Array indices range from `0` to `N-1`. This syntax is used to index into vectors, matrices, and repeated scope variables.
`expr([expr[,expr]...])`	Function call or macro expansion. The expression outside the parentheses is the function/macro name; the expressions inside are the arguments to the function or macro. **Note:** Macros are text based; they cannot be used within the same expression as other operators.
`expr . expr`	The first expression must have a valid scope; the second expression is a parameter name within that scope.
`(expr)`	Use `()` to override the precedence of operations.
`!expr`	Logical negation (generates `TLC_TRUE` or `TLC_FALSE`). The argument must be numeric or Boolean.
`-expr`	Unary minus negates the expression. The argument must be numeric.
`+expr`	No effect; the operand must be numeric.
`~expr`	Bit-wise negation of the operand. The argument must be an integer.
`expr * expr`	Multiplies the two expressions; the operands must be numeric.
`expr / expr`	Divides the two expressions; the operands must be numeric.
`expr % expr`	Takes the integer modulo of the expressions; the operands must be integers.
`expr + expr`	Works on numeric types, strings, vectors, matrices, and records as follows:

Expression	Definition
	Numeric types: Add the two expressions; the operands must be numeric. **Strings:** The strings are concatenated. **Vectors:** If the first argument is a vector and the second is a scalar, the scalar is appended to the vector. **Matrices:** If the first argument is a matrix and the second is a vector of the same column width as the matrix, the vector is appended as another row in the matrix. **Records:** If the first argument is a record, the second argument is added as a parameter identifier (with its current value). Note that the addition operator is associative.
`expr - expr`	Subtracts the two expressions; the operands must be numeric.
`expr << expr`	Left-shifts the left operand by an amount equal to the right operand; the arguments must be integers.
`expr >> expr`	Right-shifts the left operand by an amount equal to the right operand; the arguments must be integers.
`expr > expr`	Tests whether the first expression is greater than the second expression; the arguments must be numeric.
`expr < expr`	Tests whether the first expression is less than the second expression; the arguments must be numeric.
`expr >= expr`	Tests whether the first expression is greater than or equal to the second expression; the arguments must be numeric.
`expr <= expr`	Tests whether the first expression is less than or equal to the second expression; the arguments must be numeric.
`expr == expr`	Tests whether the two expressions are equal.

Expression	Definition
`expr != expr`	Tests whether the two expressions are not equal.
`expr & expr`	Performs the bit-wise AND of the two arguments; the arguments must be integers.
`expr ^ expr`	Performs the bit-wise XOR of the two arguments; the arguments must be integers.
`expr | expr`	Performs the bit-wise OR of the two arguments; the arguments must be integers.
`expr && expr`	Performs the logical AND of the two arguments and returns `TLC_TRUE` or `TLC_FALSE`. This can be used on either numeric or Boolean arguments.
`expr || expr`	Performs the logical OR of the two arguments and returns `TLC_TRUE` or `TLC_FALSE`. This can be used on either numeric or Boolean arguments.
`expr ? expr : expr`	Tests the first expression for `TLC_TRUE`. If true, the first expression is returned; otherwise, the second expression is returned.
`expr , expr`	Returns the value of the second expression.

Note Relational operators (`<, =<, >, >=, !=, ==`) can be used with nonfinite values.

You do not have to place expressions in the `%< >eval` format when they appear on directive lines. Doing so causes a double evaluation.

TLC Data Promotions

When the Target Language Compiler operates on mixed types of expressions, it promotes the results to the common types indicated in the following table.

This table uses the following abbreviations:

B	Boolean
N	Number
U	Unsigned
F	Real32
D	Real
G	Gaussian
UG	UnsignedGaussian
C32	Complex32
C	Complex

The top row (in bold) and first column (in bold) show the types of expression used in the operation. The intersection of the row and column shows the resulting type of expressions.

For example, if the operation involves a Boolean expression (B) and an unsigned expression (U), the result will be an unsigned expression (U).

Data Types Resulting from Expressions of Mixed Type

	B	**N**	**U**	**F**	**D**	**G**	**UG**	**C32**	**C**
B	B	N	U	F	D	G	UG	C32	C
N	N	N	U	F	D	G	UG	C32	C
U	U	U	U	F	D	UG	UG	C32	C
F	F	F	F	F	D	C32	C32	C32	C
D	D	D	D	D	D	C	C	C	C
G	G	G	UG	C32	C	G	UG	C32	C
UG	UG	UG	UG	C32	C	UG	UG	C32	C

	B	N	U	F	D	G	UG	C32	C
C32	C32	C32	C32	C32	C	C32	C32	C32	C
C	C	C	C	C	C	C	C	C	C

Formatting

By default, the Target Language Compiler outputs floating-point numbers in exponential notation with 16 digits of precision. To override the default, use the directive

```
%realformat string
```

If *`string`* is `"EXPONENTIAL"`, the standard exponential notation with 16 digits of precision is used. If *`string`* is `"CONCISE"`, the compiler uses internal heuristics to output the values in a more readable form while maintaining accuracy. The `%realformat` directive sets the default format for `Real` number output to the selected style for the remainder of processing or until it encounters another `%realformat` directive.

Conditional Inclusion

The conditional inclusion directives are

```
%if constant-expression
%else
%elseif constant-expression
%endif

%switch constant-expression
%case constant-expression
%break
%default
%endswitch
```

%if

The `constant-expression` must evaluate to an integer expression. It controls the inclusion of the following lines until it encounters an `%else`, `%elseif`, or `%endif` directive. If the `constant-expression` evaluates to `0`, the lines following the directive are not included. If the `constant-expression` evaluates to an integer value other

than `0`, the lines following the `%if` directive are included until the `%endif`, `%elseif`, or `%else` directive.

When the compiler encounters an `%elseif` directive, and no prior `%if` or `%elseif` directive has evaluated to nonzero, the compiler evaluates the expression. If the value is 0, the lines following the `%elseif` directive are not included. If the value is nonzero, the lines following the `%elseif` directive are included until the subsequent `%else`, `%elseif`, or `%endif` directive.

The `%else` directive begins the inclusion of source text if the previous `%elseif` statements or the original `%if` statement evaluates to `0`; otherwise, it prevents the inclusion of subsequent lines up to and including the following `%endif`.

The `constant-expression` can contain any expression specified in "Target Language Expressions" on page 6-20.

%switch

The `%switch` statement evaluates the constant expression and compares it to expressions appearing on `%case` selectors. If a match is found, the body of the `%case` is included; otherwise the `%default` is included.

`%case ... %default` bodies flow together, as in C, and `%break` must be used to exit the switch statement. `%break` exits the nearest enclosing `%switch`, `%foreach`, or `%for` loop in which it appears. For example,

```
%switch(type)
%case x
 /* Matches variable x. */
 /* Note: Any valid TLC type is allowed. */
%case "Sin"
 /* Matches Sin or falls through from case x. */
   %break
 /* Exits the switch. */
%case "gain"
 /* Matches gain. */
   %break
%default
 /* Does not match x, "Sin," or "gain." */
%endswitch
```

In general, this is a more readable form for the `%if`/`%elseif`/`%else` construction.

Multiple Inclusion

%foreach

The syntax of the `%foreach` multiple inclusion directive is

```
%foreach identifier = constant-expression
   %break
   %continue
%endforeach
```

The `constant-expression` must evaluate to an integer expression, which then determines the number of times to execute the `foreach` loop. The `identifier` increments from `0` to one less than the specified number. Within the `foreach` loop, you can use `x`, where `x` is the identifier, to access the identifier variable. `%break` and `%continue` are optional directives that you can include in the `%foreach` directive:

- Use `%break` to exit the nearest enclosing `%for`, `%foreach`, or `%switch` statement.
- Use `%continue` to begin the next iteration of a loop.

%for

Note The `%for` directive is functional, but it is not recommended. Instead, use `%roll`, which provides the same capability in a more open way. The Simulink Coder code generation software does not use the `%for` construct.

The syntax of the `%for` multiple inclusion directive is

```
%for ident1 = const-exp1, const-exp2, ident2 = const-exp3
  %body
    %break
    %continue
  %endbody
%endfor
```

The first portion of the `%for` directive is identical to the `%foreach` statement in that it causes a loop to execute from `0` to `N-1` times over the body of the loop. In the normal case, it includes only the lines between `%body` and `%endbody`, and the lines between the `%for` and `%body`, and ignores the lines between `%endbody` and `%endfor`.

The `%break` and `%continue` directives act the same as they do in the `%foreach` directive.

`const-exp2` is a Boolean expression that indicates whether the loop should be rolled. If `const-exp2` is true, `ident2` receives the value of `const-exp3`; otherwise it receives the null string. When the loop is rolled, the lines between the `%for` and the `%endfor` are included in the output exactly one time. `ident2` specifies the identifier to be used for testing whether the loop was rolled within the body. For example,

```
%for Index = <NumNonVirtualSubsystems>3, rollvar="i"

    {
        int i;

        for (i=0; i< %<NumNonVirtualSubsystems>; i++)
        {
          %body
 x[%<rollvar>] = system_name[%<rollvar>];
          %endbody

        }
    }
%endfor
```

If the number of nonvirtual subsystems (`NumNonVirtualSubsystems`) is greater than or equal to 3, the loop is rolled, causing the code within the loop to be generated exactly once. In this case, `Index = 0`.

If the loop is not rolled, the text before and after the body of the loop is ignored and the body is generated `NumNonVirtualSubsystems` times.

This mechanism gives each individual loop control over whether or not it should be rolled.

%roll

The syntax of the `%roll` multiple inclusion directive is

```
%roll ident1 = roll-vector-exp, ident2 = threshold-exp, ...
               block-exp [, type-string [,exp-list] ]
  %break
  %continue
%endroll
```

This statement uses the `roll-vector-exp` to expand the body of the `%roll` statement multiple times as in the `%foreach` statement. If a range is provided in the `roll-`

vector-expand that range is larger than the threshold-exp expression, the loop will roll. When a loop rolls, the body of the loop is expanded once and the identifier (ident2) provided for the threshold expression is set to the name of the loop control variable. If no range is larger than the specified rolling threshold, this statement is identical to the %foreach statement. For example,

```
%roll Idx = [ 1 2 3:5, 6, 7:10 ], lcv = 10, ablock
%endroll
```

In this case, the body of the %roll statement expands 10 times, as in the %foreach statement, because there are no regions greater than or equal to 10. Idx counts from 1 to 10, and lcv is set to the null string, "".

When the Target Language Compiler determines that a given block will roll, it performs a GENERATE_TYPE function call to output the various pieces of the loop (other than the body). The default type used is Roller; you can override this type with a string that you specify. Extra arguments passed to the %roll statement are provided as arguments to these special-purpose functions. The called function is one of these four functions:

RollHeader(block, ...)

This function is called once on the first section of this roll vector that will actually roll. It should return a string that is assigned to the lcv within the body of the %roll statement.

LoopHeader(block, StartIdx, Niterations, Nrolled, ...)

This function is called once for each section that will roll prior to the body of the %roll statement.

LoopTrailer(block, Startidx, Niterations, Nrolled, ...)

This function is called once for each section that will roll after the body of the %roll statement.

RollTrailer(block, ...)

This function is called once at the end of the %roll statement if any of the ranges caused loop rolling.

These functions should output language-specific declarations, loop code, and so on as required to generate code for the loop.

An example of a Roller.tlc file is

```
%implements Roller "C"
%function RollHeader(block) Output
 {
  int i;
 %return ("i")
%endfunction
%function LoopHeader(block,StartIdx,Niterations,Nrolled) Output
  for (i = %<StartIdx>; i < %<Niterations+StartIdx>; i++)
 {
%endfunction
%function LoopTrailer(block,StartIdx,Niterations,Nrolled) Output
 }
%endfunction
%function RollTrailer(block) Output
 }
%endfunction
```

Note The Target Language Compiler function library provided with the Simulink Coder product has the capability to extract references to the block I/O and other Simulink Coder vectors that vastly simplify the body of the `%roll` statement. These functions include `LibBlockInputSignal`, `LibBlockOutputSignal`, `LibBlockParameter`, `LibBlockRWork`, `LibBlockIWork`, `LibBlockPWork`, `LibDeclareRollVars`, `LibBlockMatrixParameter`, `LibBlockParameterAddr`, `LibBlockContinuousState`, and `LibBlockDiscreteState`. (See the function reference pages in "Input Signal Functions" on page 9-8, "Output Signal Functions" on page 9-20, "Parameter Functions" on page 9-27, and "Block State and Work Vector Functions" on page 9-35.) This library also includes a default implementation of `Roller.tlc` as a "flat" roller.

Extending the former example to a loop that rolls,

```
%language "C"
%assign ablock = BLOCK { Name "Hi" }
%roll Idx = [ 1:20, 21, 22, 23:25, 26:46], lcv = 10, ablock
 Block[%< lcv == "" ? Idx : lcv>] *= 3.0;
%endroll
```

This Target Language Compiler code produces this output:

```
{
 int             i;
 for (i = 1; i < 21; i++)
```

```
 {
     Block[i] *= 3.0;
 }
 Block[21] *= 3.0;
 Block[22] *= 3.0;
 Block[23] *= 3.0;
 Block[24] *= 3.0;
 Block[25] *= 3.0;
 for (i = 26; i < 47; i++)
 {
     Block[i] *= 3.0;
 }
}
```

Object-Oriented Facility for Generating Target Code

The Target Language Compiler provides a simple object-oriented facility. The language directives are

```
%language string
%generatefile
%implements
```

This facility was designed specifically for customizing the code for Simulink blocks, but can be used for other purposes as well.

%language

The `%language` directive specifies the target language being generated. It is required as a consistency check to verify the implementation files found for the language being generated. The `%language` directive must appear prior to the first `GENERATE` or `GENERATE_TYPE` built-in function call. `%language` specifies the language as a string. For example:

```
%language "C"
```

Simulink blocks have a `Type` parameter. This parameter is a string that specifies the type of the block, for example `"Sin"` or `"Gain"`. The object-oriented facility uses this type to search the path for a file that implements the block. By default the name of the file is the `Type` of the block with `.tlc` appended, so for example, if the `Type` is `"Sin"` the Compiler would search for `"Sin.tlc"` along the path. You can override this default filename using the `%generatefile` directive to specify the filename that you want to use to replace the default filename. For example,

```
%generatefile "Sin" "sin_wave.tlc"
```

The files that implement the block-specific code must contain an `%implements` directive indicating both the type and the language being implemented. The Target Language Compiler will produce an error if the `%implements` directive does not match as expected. For example,

```
%implements "Sin" "Pascal"
```

causes an error if the initial language choice was C.

You can use a single file to implement more than one target language by specifying the desired languages in a vector. For example,

```
%implements "Sin" "C"
```

Finally, you can implement several types using the wildcard (`*`) for the type field:

```
%implements * "C"
```

Note The use of the wildcard (`*`) is not recommended because it relaxes error checking for the `%implements` directive.

GENERATE and GENERATE_TYPE Functions

The Target Language Compiler has two built-in functions that dispatch object-oriented calls, `GENERATE` and `GENERATE_TYPE`. You can call a function appearing in an implementation file (from outside the specified file) only by using the `GENERATE` and `GENERATE_TYPE` special functions.

GENERATE

The `GENERATE` function takes two or more input arguments. The first argument must be a valid scope and the second a string containing the name of the function to call. The `GENERATE` function passes the first block argument and any additional arguments specified to the function being called. The return argument is the value, if any, returned from the function being called. Note that the compiler automatically "scopes" or adds the first argument to the list of scopes searched as if it appears on a `%with` directive line. (See "Variable Scoping" on page 6-50.) This scope is removed when the function returns.

GENERATE_TYPE

The GENERATE_TYPE function takes three or more input arguments. It handles the first two arguments identically to the GENERATE function call. The third argument is the type; the type specified in the Simulink block is ignored. This facility is used to handle S-function code generation by the Simulink Coder build process. That is, the block type is S-function, but the Target Language Compiler generates it as the specific S-function specified by GENERATE_TYPE. For example,

```
GENERATE_TYPE(block, "Output", "dp_read")
```

specifies that S-function block is of type dp_read.

The block argument and any additional arguments are passed to the function being called. Like the GENERATE built-in function, the compiler automatically scopes the first argument before the GENERATE_TYPE function is entered and then removes the scope on return.

Within the file containing %implements, function calls are looked up first within the file and then in the global scope. This makes it possible to have hidden helper functions used exclusively by the current object.

Note It is not an error for the GENERATE and GENERATE_TYPE directives to find no matching functions. This is to prevent requiring empty specifications for all aspects of block code generation. Use the GENERATE_FUNCTION_EXISTS or GENERATE_TYPE_FUNCTION_EXISTS directives to determine whether the specified function actually exists.

Output File Control

The structure of the output file control construct is

```
%openfile string optional-equal-string optional-mode
%closefile id
%selectfile id
```

%openfile

The `%openfile` directive opens a file or buffer for writing; the required string variable becomes a variable of type `file`. For example,

```
%openfile x               /% Opens and selects x for writing. %/
%openfile out = "out.h"   /% Opens "out.h" for writing. %/
```

%selectfile

The `%selectfile` directive selects the file specified by the variable as the current output stream. Output goes to that file until another file is selected using `%selectfile`. For example,

```
%selectfile x               /% Select file x for output. %/
```

%closefile

The `%closefile` directive closes the specified file or buffer. If the closed entity is the currently selected output stream, `%closefile` invokes `%selectfile` to reselect the previously selected output stream.

There are two possible cases that `%closefile` must handle:

- If the stream is a file, the associated variable is removed as if by `%undef`.
- If the stream is a buffer, the associated variable receives the text that has been output to the stream. For example,

  ```
  %assign x = ""                /% Creates an empty string. %/
  %openfile x
  "hello, world"
  %closefile x   /% x = "hello, world\n"%/
  ```

If desired, you can append to an output file or string by using the optional mode, `a`, as in

```
%openfile "foo.c", "a"             %% Opens foo.c for appending.
```

Input File Control

The input file control directives are

```
%include string
%addincludepath string
```

%include

The `%include` directive searches the path for the target file specified by `string` and includes the contents of the file inline at the point where the `%include` statement appears.

%addincludepath

The `%addincludepath` directive adds an additional include path to be searched when the Target Language Compiler references `%include` or block target files. The syntax is

```
%addincludepath string
```

The `string` can be an absolute path or an explicit relative path. For example, to specify an absolute path, use

```
%addincludepath "C:\\folder1\\folder2"     (PC)
%addincludepath "/folder1/folder2"         (UNIX)
```

To specify a relative path, the path must explicitly start with `.`. For example,

```
%addincludepath ".\\folder2"                  (PC)
%addincludepath "./folder2"                   (UNIX)
```

Note that for PC, the backslashes must be escaped (doubled).

When an explicit relative path is specified, the folder that is added to the Target Language Compiler search path is created by concatenating the location of the target file that contains the `%addincludepath` directive and the explicit relative path.

The Target Language Compiler searches the folders in the following order for target or include files:

1. The current folder.
2. Include paths specified in `%addincludepath` directives. The compiler evaluates multiple `%addincludepath` directives from the *bottom up*.

3 Include paths specified at the command line via `-I` . The compiler evaluates multiple `-I` options from *right to left*.

Typically, `%addincludepath` directives should be specified in your system target file. Multiple `%addincludepath` directives will add multiple paths to the Target Language Compiler search path.

Note: The compiler does *not* search the MATLAB path, and will not find a file that is available only on that path. The compiler searches only the locations described above.

Asserts, Errors, Warnings, and Debug Messages

The related assert, error, warning, and debug message directives are

```
%assert expression
%error tokens
%warning tokens
%trace tokens
%exit tokens
```

These directives produce error, warning, or trace messages whenever a target file detects an error condition, or tracing is desired. The tokens following the directive on a line become part of the generated error or warning message.

The Target Language Compiler places messages generated by `%trace` onto `stderr` if and only if you specify the verbose mode switch (`-v`) to the Target Language Compiler. See “Command-Line Arguments” on page 6-64 for additional information about switches.

The `%assert` directive evaluates the expression and produces a stack trace if the expression evaluates to a Boolean `false`.

Note In order for `%assert` directives to be evaluated, **Enable TLC assertion** must be selected in the **TLC process** section of the **Code Generation** > **Debug** pane. The default action is for `%assert` directives not to be evaluated.

The `%exit` directive reports an error and stops further compilation.

Built-In Functions and Values

The following table lists the built-in functions and values that are added to the list of parameters that appear in the *model*.rtw file. These Target Language Compiler functions and values are defined in uppercase so that they are visually distinct from other parameters in the *model*.rtw file, and, by convention, from user-defined parameters.

TLC Built-In Functions and Values

Built-In Function Name	Expansion
`CAST(expr, expr)`	The first expression must be a string that corresponds to one of the type names in the table "Target Language Value Types" on page 6-18, and the second expression will be cast to that type. A typical use might be to cast a variable to a real format as in `CAST("Real", variable-name)` An example of this is in working with parameter values for S-functions. To use them within C or C++ code, you need to typecast them to `real` so that a value such as `1` will be formatted as `1.0` (see also `%realformat`).
`EXISTS(var)`	If the `var` identifier is not currently in scope, the result is `TLC_FALSE`. If the identifier is in scope, the result is `TLC_TRUE`. `var` can be a single identifier or an expression involving the `.` and `[]` operators.

`FEVAL(matlab-command, TLC-expressions)`	Performs an evaluation in MATLAB. For example,
	`%assign result = FEVAL("sin",3.14159)`
	The `%matlab` directive can be used to call a MATLAB function that does not return a result. For example,
	`%matlab disp(2.718)`
	Note: If the MATLAB function returns more than one value, TLC receives the first value only.

Built-In Function Name	Expansion
`FILE_EXISTS(expr)`	`expr` must be a string. If a file by the name `expr` does not exist on the path, the result is `TLC_FALSE`. If a file by that name exists on the path, the result is `TLC_TRUE`.
`FORMAT(realvalue, format)`	The first expression is a `Real` value to format. The second expression is either `EXPONENTIAL` or `CONCISE`. Outputs the `Real` value in the designated format, where `EXPONENTIAL` uses exponential notation with 16 digits of precision, and `CONCISE` outputs the number in a more readable format while maintaining numerical accuracy.
`FIELDNAMES(record)`	Returns an array of strings containing the record field names associated with the record. Because it returns a sorted list of strings, the function is `O(n*log(n))`.
`GETFIELD(record, "fieldname")`	Returns the contents of the specified field name, if the field name is associated with the record. The function uses hash lookup, and therefore executes in constant time.
`GENERATE(record, function-name, ...)`	Executes function calls mapped to a specific record type (i.e., block record functions). For example, use this to execute the functions in the `.tlc` files for built-in blocks. Not that TLC automatically "scopes" or adds the first argument to the list of scopes searched as if it appears on a `%with` directive line.
`GENERATE_FILENAME(type)`	For the specified record type, does a `.tlc` file exist? Use this to see if the `GENERATE_TYPE` call will succeed.
`GENERATE_FORMATTED_VALUE (expr, string, expand)`	Returns a potentially multiline string that can be used to declare the value(s) of `expr` in the current target language. The second argument is a string that is used as the variable name in a descriptive comment on the first line of the return string. If the second argument is the empty string, `""`, then no descriptive comment is put into the output string. The third argument is a Boolean that when `TRUE` causes `expr` to be expanded into raw text before being output. `expand = TRUE` uses much more memory than the default (`FALSE`); set `expand = TRUE` only if the parameter text needs to be processed for some reason before being written to disk.
`GENERATE_FUNCTION_EXISTS (record, function-name)`	Determines whether a given block function exists. The first expression is the same as the first argument to `GENERATE`,

Built-In Function Name	Expansion
	namely a block scoped variable containing a `Type`. The second expression is a string that should match the function name.
`GENERATE_TYPE (record, function-name, type, ...)`	Similar to `GENERATE`, except that `type` overrides the `Type` field of the record. Use this when executing functions mapped to specific S-function block records based upon the S-function name (i.e., the name becomes the type).
`GENERATE_TYPE_FUNCTION_EXIS (record, function-name, type)`	Same as `GENERATE_FUNCTION_EXISTS` except that it overrides the `Type` built into the record.
`GET_COMMAND_SWITCH`	Returns the values of command-line switches. Only the following switches are supported: `v, m, p, O, d, dr, r, I, a` See also "Command-Line Arguments" on page 6-64.
`IDNUM(expr)`	`expr` must be a string. The result is a vector where the first element is a leading string, if any, and the second element is a number appearing at the end of the input string. For example, `IDNUM("ABC123")` yields `["ABC", 123]`
`IMAG(expr)`	Returns the imaginary part of a complex number.
`INT8MAX`	`127`
`INT8MIN`	`-128`
`INT16MAX`	`32767`
`INT16MIN`	`-32768`
`INT32MAX`	`2147483647`
`INT32MIN`	`-2147483648`
`INTMIN`	Minimum integer value on host machine.
`INTMAX`	Maximum integer value on host machine.
`ISALIAS(record)`	Returns `TLC_TRUE` if the record is a reference (symbolic link) to a different record, and `TLC_FALSE` otherwise.
`ISEQUAL(expr1, expr2)`	Where the data types of both expressions are numeric: returns `TLC_TRUE` if the first expression contains the same value as the second expression; returns `TLC_FALSE` otherwise.

Built-In Function Name	Expansion
	Where the data type of either expression is nonnumeric (e.g., string or record): returns TLC_TRUE if and only if both expressions have the same data type and contain the same value; returns TLC_FALSE otherwise.
ISEMPTY(expr)	Returns TLC_TRUE if the expression contains an empty string, vector, or record, and TLC_FALSE otherwise.
ISFIELD(record, "fieldname")	Returns TLC_TRUE if the field name is associated with the record, and TLC_FALSE otherwise.
ISINF(expr)	Returns TLC_TRUE if the value of the expression is inf, and TLC_FALSE otherwise.
ISNAN(expr)	Returns TLC_TRUE if the value of the expression is NAN, and TLC_FALSE otherwise.
ISFINITE(expr)	Returns TLC_TRUE if the value of the expression is not +/- inf or NAN, and TLC_FALSE otherwise.

ISSLPRMREF(param.value)	Returns a Boolean value indicating whether its argument is a reference to a Simulink parameter or not. This function supports parameter sharing with Simulink; using it can save memory and time during code generation. For example,
	
	`%if !ISSLPRMREF(param.Value)`
	`  assign param.Value = CAST("Real", param.Value)`
	`%endif`

NULL_FILE	A predefined file for no output that you can use as an argument to %selectfile to prevent output.
NUMTLCFILES	The number of target files used thus far in expansion.
OUTPUT_LINES	Returns the number of lines that have been written to the currently selected file or buffer. Does not work for STDOUT or NULL_FILE.
REAL(expr)	Returns the real part of a complex number.
REMOVEFIELD(record, "fieldname")	Removes the specified field from the contents of the record. Returns TLC_TRUE if the field was removed; otherwise returns TLC_FALSE.
ROLL_ITERATIONS()	Returns the number of times the current roll regions are looping or NULL if not inside a %roll construct.

Built-In Function Name	Expansion
`SETFIELD(record, "fieldname", value)`	Sets the contents of the field name associated with the record. Returns `TLC_TRUE` if the field was added; otherwise returns `TLC_FALSE`.
`SIZE(expr[,expr])`	Calculates the size of the first expression and generates a two-element row vector. If the second operand is specified, it is used as an integer index into this row vector; otherwise the entire row vector is returned. `SIZE(x)` applied to a scalar returns `[1 1]`. `SIZE(x)` applied to a scope returns the number of repeated entries of that scope type. For example, `SIZE(Block)` returns `[1,<number of blocks>]`
`SPRINTF(format,var,...)`	Formats the data in variable `var` (and in any additional variable arguments) under control of the specified format string, and returns a string variable containing the values. Operates like the C library `sprintf()`, except that output is the return value rather than contained in an argument to `sprintf`.
`STDOUT`	A predefined file for `stdout` output. You can use this as an argument to `%selectfile` to force output to `stdout`.
`STRING(expr)`	Expands the expression into a string; the characters `\`, `\n`, and `"` are escaped by preceding them with `\` (backslash). The ANSI escape sequences are translated into string form. If `%<>` is in the expression, it is escaped so that the enclosed string is not subject to further TLC interpretation.
`STRINGOF(expr)`	Accepts a vector of ASCII values and returns a string that is constructed by treating each element as the ASCII code for a single character. Used primarily for S-function string parameters.
`SYSNAME(expr)`	Looks for specially formatted strings of the form `<x>/y` and returns `x` and `y` as a two-element string vector. This is used to resolve subsystem names. For example, `%<sysname("<sub>/Gain")>` returns

Built-In Function Name	Expansion
	`["sub","Gain"]` In `Block` records, the name of the block is written `<sys/blockname>`, where *sys* is `S#` or `Root`. You can obtain the full path name by calling `LibGetBlockPath(block)`; this will include newlines and other troublesome characters that cause display difficulties. To obtain a full path name suitable for one-line comments but not identical to the Simulink path name, use `LibGetFormattedBlockPath(block)`.
`TLCFILES`	Returns a vector containing the names of the target files included thus far in the expansion. Absolute paths are used. See also `NUMTLCFILES`.
`TLC_FALSE`	Boolean constant that equals a negative evaluated Boolean expression.
`TLC_TRUE`	Boolean constant that equals a positive evaluated Boolean expression.
`TLC_TIME`	Date and time of compilation.
`TLC_VERSION`	Version and date of the Target Language Compiler.
`TYPE(expr)`	Evaluates `expr` and determines the result type. The result of this function is a string that corresponds to the type of the given expression. See the **Value Type String** column in the table "Target Language Value Types" on page 6-18 for possible values.
`UINT8MAX`	`255U`
`UINT16MAX`	`65535U`
`UINT32MAX`	`4294967295U`
`UINTMAX`	Maximum unsigned integer value on host machine.
`WHITE_SPACE(expr)`	Accepts a string and returns 1 if the string contains only white-space characters (, `\t`, `\n`, `\r`); returns 0 otherwise.
`WILL_ROLL(expr1, expr2)`	The first expression is a roll vector and the second expression is a roll threshold. This function returns `true` if the vector contains a range that will roll.

FEVAL Function

The FEVAL built-in function calls MATLAB file functions and MEX-functions. The structure is

```
%assign result = FEVAL( matlab-function-name, rhs1, rhs2, ...
 rhs3, ... );
```

Note Only a single left-side argument is allowed when you use FEVAL.

This table shows the conversions that are made when you use FEVAL.

MATLAB Conversions

TLC Type	MATLAB Type
"Boolean"	Boolean (scalar or matrix)
"Number"	Double (scalar or matrix)
"Real"	Double (scalar or matrix)
"Real32"	Double (scalar or matrix)
"Unsigned"	Double (scalar or matrix)
"String"	String
"Vector"	If the vector is homogeneous, it is converted to a MATLAB vector. If the vector is heterogeneous, it is converted to a MATLAB cell array.
"Gaussian"	Complex (scalar or matrix)
"UnsignedGaussian"	Complex (scalar or matrix)
"Complex"	Complex (scalar or matrix)
"Complex32"	Complex (scalar or matrix)
"Identifier"	String
"Subsystem"	String
"Range"	Expanded vector of Doubles
"Idrange"	Expanded vector of Doubles

TLC Type	MATLAB Type
`"Matrix"`	If the matrix is homogeneous, it is converted to a MATLAB matrix. If the matrix is heterogeneous, it is converted to a MATLAB cell array. (Cell arrays can be nested.)
`"Scope"` or `"Record"`	Structure with elements
`Scope` or `Record` alias	String containing fully qualified alias name
`Scope` or `Record` array	Cell array of structures
Other type not listed above	Conversion not supported

When values are returned from MATLAB, they are converted as shown in this table. Note that conversion of matrices with more than two dimensions is not supported, nor is conversion or downcast of 64-bit integer values.

More Conversions

MATLAB Type	TLC Type
`String`	`String`
`Vector of Strings`	Vector of Strings
`Boolean (scalar or matrix)`	`Boolean` (scalar or matrix)
`INT8,INT16,INT32 (scalar or matrix)`	`Number` (scalar or matrix)
`INT64`	Not supported
`UINT64`	Not supported
`Complex INT8,INT16,INT32 (scalar or matrix)`	Gaussian (scalar or matrix)
`UINT8,UINT16,UINT32 (scalar or matrix)`	`Unsigned` (scalar or matrix)
`Complex UINT8,UINT16,UINT32 (scalar or matrix)`	`UnsignedGaussian` (scalar or matrix)
`Single precision`	`Real32` (scalar or matrix)
`Complex single precision`	`Complex32` (scalar or matrix)
`Double precision`	`Real` (scalar or matrix)

MATLAB Type	TLC Type
`Complex double precision`	`Complex` (scalar or matrix)
`Sparse matrix`	Expanded to matrix of `Doubles`
`Cell array of structures`	Record array
`Cell array of non-structures`	Vector or matrix of types converted from the types of the elements
`Cell array of structures and non-structures`	Conversion not supported
`Structure`	`Record`
`Object`	Conversion not supported

Other value types are not currently supported.

As an example, this statement uses the `FEVAL` built-in function to call MATLAB to take the sine of the input argument.

```
%assign result = FEVAL( "sin", 3.14159 )
```

Variables (identifiers) can take on the following constant values. Note the suffix on the value .

Constant Form	TLC Type
`1.0`	`"Real"`
`1.0[F/f]`	`"Real32"`
`1`	`"Number"`
`1[U|u]`	`"Unsigned"`
1.0i	`"Complex"`
`1[Ui|ui]`	`"UnsignedGaussian"`
`1i`	`"Gaussian"`
`1.0[Fi|fi]`	`"Complex32"`

Note The suffix controls the Target Language Compiler type obtained from the constant.

This table shows Target Language Compiler constants and their equivalent MATLAB values.

TLC Constants	Equivalent MATLAB Value
`rtInf`, `Inf`, `inf`	`+inf`
`rtMinusInf`	`-inf`
`rtNan`, `NaN`, `nan`	`nan`
`rtInfi`, `Infi`, `infi`	`inf*i`
`rtMinusInfi`	`-inf*i`
`rtNaNi`, `NaNi`, `nani`	`nan*i`

TLC Reserved Constants

For double-precision values, the following are defined for infinite and not-a-number IEEE® values:

```
rtInf, inf, rtMinusInf, -inf, rtNaN, nan
```

For single-precision values, these constants apply:

```
rtInfF, InfF, rtMinusInfF, rtNaNF, NaNF
```

Their corresponding versions when complex are:

```
rtInfi, infi, rtMinusInfi, -infi, rtNaNi (for doubles)
rtInfFi, InfFi, rtMinusInfFi, rtNaNFi, NaNFi (for singles)
```

For integer values, the following are defined:

```
INT8MIN, INT8MAX, INT16MIN, INT16MAX, INT32MIN, INT32MAX,
UINT8MAX, UINT16MAX, UINT32MAX, INTMAX, INTMIN, UINTMAX
```

Identifier Definition

To define or change identifiers (TLC variables), use the directive

```
%assign [::]expression = constant-expression
```

This directive introduces new identifiers (variables) or changes the values of existing ones. The left side can be a qualified reference to a variable using the `.` and `[]` operators,

or it can be a single element of a vector or matrix. In the case of the matrix, only the single element is changed by the assignment.

The `%assign` directive inserts new identifiers into the local function scope, file function scope, generate file scope, or into the global scope. Identifiers introduced into the function scope are not available within functions being called, and are removed upon return from the function. Identifiers inserted into the global scope are persistent. You can change existing identifiers by completely respecifying them. The constant expressions can include legal identifiers from the `.rtw` files. You can use `%undef` to delete identifiers in the same way that you use it to remove macros.

Within the scope of a function, variable assignments create new local variables unless you use the `::` scope resolution operator. For example, given a local variable `foo` and a global variable `foo`,

```
%function ...
...
%assign foo = 3
...
%endfunction
```

In this example, the assignment creates a variable `foo`, local to the function, that will disappear when the function exits. Note that `foo` is created even if a global `foo` already exists.

To create or change values in the global scope, you must use the scope resolution operator (`::`) to disambiguate, as in

```
%function ...
%assign foo = 3
%assign ::foo = foo
...
%endfunction
```

The scope resolution operator `(` forces the compiler to assign to the global `foo`, or to change its existing value to 3.

Note It is an error to change a value from the Simulink Coder file without qualifying it with the scope. This example does not generate an error:

```
 %assign CompiledModel.name = "newname" %% No error
```

This example generates an error:

```
%with CompiledModel
  %assign name = "newname"    %% Error  %endwith
```

Creating Records

Use the `%createrecord` directive to build new records in the current scope. For example, if you want to create a new record called `Rec` that contains two items (e.g., `Name "Name"` and `Type "t"`), use

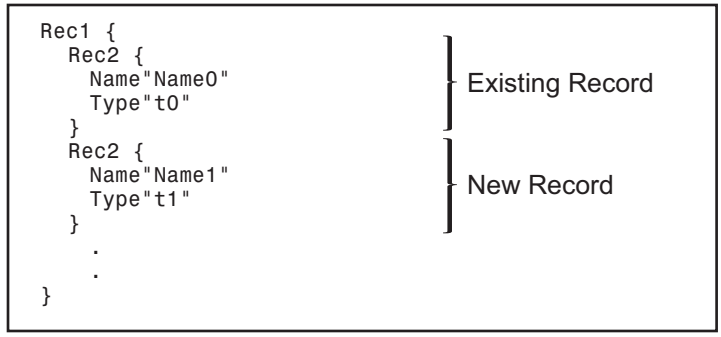
```
%createrecord Rec {  Name "Name"; Type "t" }
```

Adding Records

Use the `%addtorecord` directive to add new records to existing records. For example, if you have a record called `Rec1` that contains a record called `Rec2`, and you want to add an additional `Rec2` to it, use

```
%addtorecord Rec1 Rec2 {  Name "Name1"; Type "t1" }
```

This figure shows the result of adding the record to the existing one.



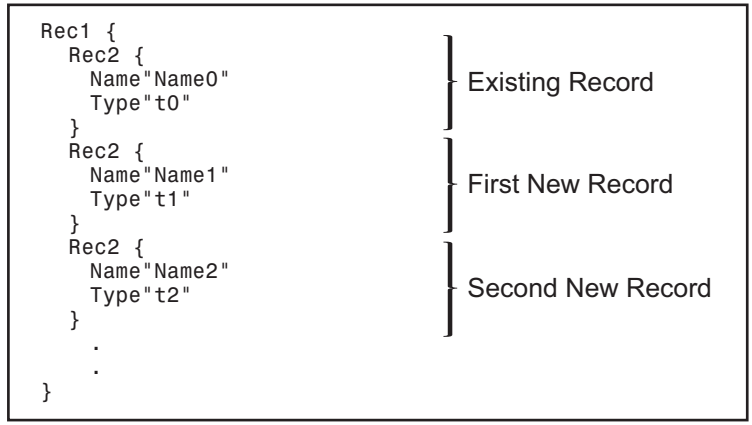
If you want to access the new record, you can use

```
%assign myname = Rec1.Rec2[1].Name
```

In this same example, if you want to add two records to the existing record, use

```
%addtorecord Rec1 Rec2 {  Name "Name1"; Type "t1" }
%addtorecord Rec1 Rec2 {  Name "Name2"; Type "t2" }
```

This produces

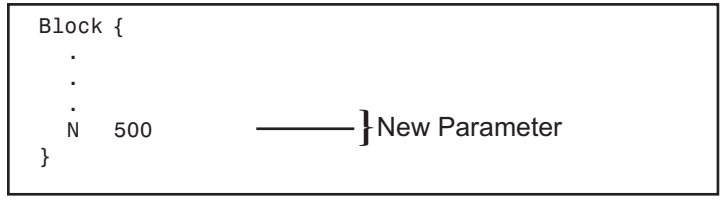


Adding Parameters to an Existing Record

You can use the `%assign` directive to add a new parameter to an existing record. For example,

```
%addtorecord Block[Idx] N 500 /% Adds N with value 500 to Block %/
%assign myn = Block[Idx].N    /% Gets the value 500 %/
```

adds a new parameter, `N`, at the end of an existing block with the name and current value of an existing variable, as shown in this figure. It returns the block value.



Variable Scoping

This section discusses how the Target Language Compiler resolves references to variables (including records).

Scope, in this document, has two related meanings. First, scope is an attribute of a variable that defines its visibility and persistence. For example, a variable defined within the body of a function is visible only within that function, and it persists only as long as that function is executing. Such a variable has *function (or local) scope*. Each TLC variable has one (and only one) of the scopes described in "Scopes" on page 6-51.

The term scope also refers to a collection, or *pool*, of variables that have the same scope. At a given point in the execution of a TLC program, several scopes can exist. For example, during execution of a function, a function scope (the pool of variables local to the function) exists. In all cases, a global scope (the pool of global variables) also exists.

To resolve variable references, TLC maintains a search list of current scopes and searches them in a well-defined sequence. The search sequence is described in "How TLC Resolves Variable References" on page 6-56.

Dynamic scoping refers to the process by which TLC creates and deallocates variables and the scopes in which they exist. For example, variables in a function scope exist only while the defining function executes.

Scopes

The following sections describe the possible scopes that a TLC variable can have.

Global Scope

By default, TLC variables have global scope. Global variables are visible to, and can be referenced by, code anywhere in a TLC program. Global variables persist throughout the execution of the TLC program. Global variables are said to belong to the *global pool*.

Note that the `CompiledModel` record of the *`model`*`.rtw` file has global scope. Therefore, you can access this structure from your TLC functions or files.

You can use the scope resolution operator (`::`) to explicitly reference or create global variables from within a function. See "The Scope Resolution Operator" on page 6-55 for examples.

Note that you can use the `%undef` directive to free memory used by global variables.

File Scope

Variables with file scope are visible only within the file in which they are created. To limit the scope of variables in this way, use the `%filescope` directive anywhere in the defining file.

In the following code fragment, the variables `fs1` and `fs2` have file scope. Note that the `%filescope` directive does not have to be positioned before the statements that create the variables.

```
%assign fs1 = 1
%filescope
%assign fs2 = 3
```

Variables whose scope is limited by `%filescope` go out of scope when execution of the file containing them completes. This lets you free memory allocated to such variables.

Function (Local) Scope

Variables defined within the body of a function have function scope. That is, they are visible within and local to the defining function. For example, in the following code fragment, the variable `localv` is local to the function `foo`. The variable `x` is global.

```
%assign x = 3

%function foo(arg)
   %assign localv = 1
   %return x + localv
%endfunction
```

A local variable can have the same name as a global variable. To refer, within a function, to identically named local and global variables, you must use the scope resolution operator (`::`) to disambiguate the variable references. See "The Scope Resolution Operator" on page 6-55 for examples.

Note Functions themselves (as opposed to the variables defined within functions) have global scope. There is one exception: functions defined in generate scope are local to that scope. See "Generate Scope" on page 6-53.

%with Scope

The `%with` directive adds a new scope, referred to as a *`%with scope`*, to the current list of scopes. This directive makes it easier to refer to block-scoped variables.

The structure of the `%with` directive is

```
%with expression
%endwith
```

For example, the directive

```
%with CompiledModel.System[sysidx]
   ...
%endwith
```

adds the `CompiledModel.System[sysidx]` scope to the search list. This scope is searched before anything else. You can then refer to the system name simply by

```
Name
```

instead of

```
CompiledModel.System[sysidx].Name
```

Generate Scope

Generate scope is a special scope used by certain built-in functions that are designed to support code generation. These functions dispatch function calls that are mapped to a specific record type. This capability supports a type of polymorphism in which different record types are associated with functions (analogous to methods) of the same name. Typically, this feature is used to map `Block` records to functions that implement the functionality of different block types.

Functions that employ generate scope include `GENERATE`, `GENERATE_TYPE`, `GENERATE_FUNCTION_EXISTS`, and `GENERATE_TYPE_FUNCTION_EXISTS`. See "GENERATE and GENERATE_TYPE Functions" on page 6-33. This section discusses generate scope using the `GENERATE` built-in function as an example.

The syntax of the `GENERATE` function is

```
GENERATE(blk,fn)
```

The first argument (`blk`) to `GENERATE` is a valid record name. The second argument (`fn`) is the name of a function to be dispatched. When a function is dispatched through a `GENERATE` call, TLC automatically adds `blk` to the list of scopes that is searched when variable references are resolved. Thus the record (`blk`) is visible to the dispatched function as if an implicit `%with <blk>... %endwith` directive existed in the dispatched function.

In this context, the record named `blk` is said to be in generate scope.

Three TLC files, illustrating the use of generate scope, are listed below. The file `polymorph.tlc` creates two records representing two hypothetical block types, `MyBlock` and `YourBlock`. Each record type has an associated function named `aFunc`. The block-specific implementations of `aFunc` are contained in the files `MyBlock.tlc` and `YourBlock.tlc`.

Using `GENERATE` calls, `polymorph.tlc` dispatches to a function for each block type. Notice that the `aFunc` implementations can refer to the fields of `MyBlock` and `YourBlock`, because these records are in generate scope.

- The following listing shows `polymorph.tlc`:

```
%% polymorph.tlc

%language "C"

%%create records used as scopes within dispatched functions

%createrecord MyRecord { Type "MyBlock"; data 123 }
%createrecord YourRecord { Type "YourBlock"; theStuff 666 }

%% dispatch the functions thru the GENERATE call.

%% dispatch to MyBlock implementation
%<GENERATE(MyRecord, "aFunc")>

%% dispatch to YourBlock implementation
%<GENERATE(YourRecord, "aFunc")>

%% end of polymorph.tlc
```

- The following listing shows `MyBlock.tlc`:

```
%%MyBlock.tlc

%implements "MyBlock" "C"

%% aFunc is invoked thru a GENERATE call in polymorph.tlc.
%% MyRecord is in generate scope in this function.
%% Therefore, fields of MyRecord can be referenced without
%% qualification

%function aFunc(r) Output
%selectfile STDOUT
The value of MyRecord.data is: %<data>
%closefile STDOUT
%endfunction

%%end of MyBlock.tlc
```

- The following listing shows `YourBlock.tlc`:

```
%%YourBlock.tlc

%implements "YourBlock" "C"
```

```
%% aFunc is invoked thru a GENERATE call in polymorph.tlc.
%% YourRecord is in generate scope in this function.
%% Therefore, fields of YourRecord can be referenced without
%% qualification

%function aFunc(r) Output
%selectfile STDOUT
The value of YourRecord.theStuff is: %<theStuff>
%closefile STDOUT
%endfunction

%%end of YourBlock.tlc
```

The invocation and output of `polymorph.tlc`, as displayed in the MATLAB Command Window, are shown below:

```
tlc -v polymorph.tlc

The value of MyRecord.data is: 123
The value of YourRecord.theStuff is: 666
```

Note Functions defined in generate scope are local to that scope. This is an exception to the general rule that functions have global scope. In the above example, for instance, neither of the `aFunc` implementations has global scope.

The Scope Resolution Operator

The scope resolution operator (`::`) is used to indicate that the global scope should be searched when a TLC function looks up a variable reference. The scope resolution operator is often used to change the value of global variables (or even create global variables) from within functions.

By using the scope resolution operator, you can resolve ambiguities that arise when a function references identically named local and global variables. In the following example, a global variable `foo` is created. In addition, the function `myfunc` creates and initializes a local variable named `foo`. The function `myfunc` explicitly references the global variable `foo` by using the scope resolution operator.

```
%assign foo = 3   %% this variable has global scope
.
.
.
```

```
%function myfunc(arg)
   %assign foo = 3   %% this variable has local scope
   %assign ::foo = arg   %% this changes the global variable foo
%endfunction
```

You can also use the scope resolution operator within a function to create global variables. The following function creates and initializes a global variable:

```
%function sideffect(arg)
   %assign ::theglobal = arg   %% this creates a global variable
%endfunction
```

How TLC Resolves Variable References

This section discusses how the Target Language Compiler searches the existing scopes to resolve variable references.

Global Scope

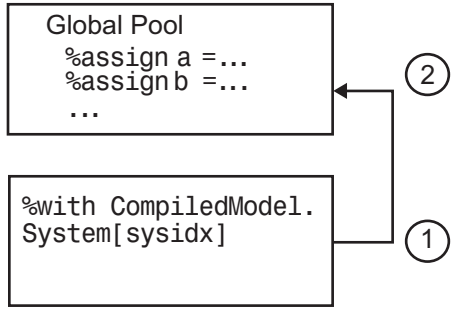
In the simplest case, the Target Language Compiler resolves a variable reference by searching the global pool (including the `CompiledModel` structure).

%with Scope

You can modify the search list and search sequence by using the `%with` directive. For example, when you add the following construct,

```
%with CompiledModel.System[sysidx]
   ...
%endwith
```

the `System[sysidx]` scope is added to the search list. This scope is searched first, as shown by this picture.



This technique makes it simpler to access embedded definitions. Using the `%with` construct (as in the previous example), you can refer to the system name simply by

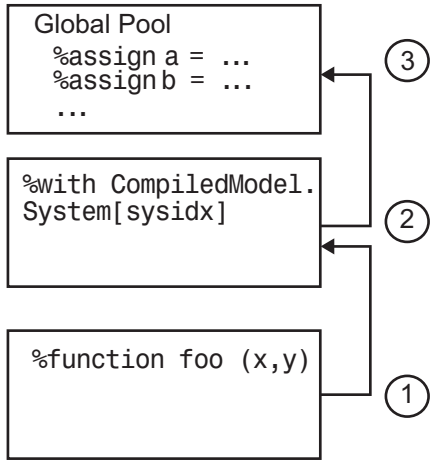
```
Name
```

instead of

```
CompiledModel.System[sysidx].Name
```

Function Scope

A function has its own scope. That scope is added to the previously described search list, as shown in this diagram.

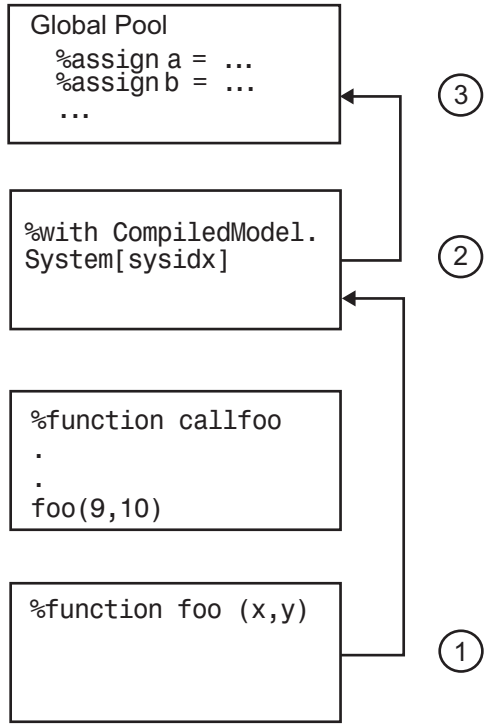


For example, in the following code fragment,

```
% with CompiledModel.System[sysidx]
...
   %assign a=foo(x,y)
...
%endwith
...
%function foo(a,b)
...
   assign myvar=Name
...
%endfunction
...
%<foo(1,2)>
```

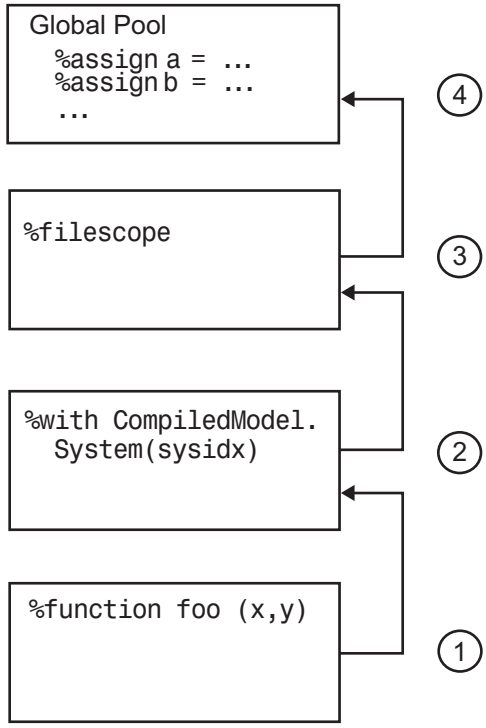
If `Name` is not defined in `foo`, the assignment uses the value of `Name` from the previous scope, `CompiledModel.System[SysIdx].Name`.

In a nested function, only the innermost function scope is searched, together with the enclosing `%with` and global scopes, as shown in the following diagram:



File Scope

File scopes are searched before the global scope, as shown in the following diagram.



The rule for nested file scopes is similar to that for nested function scopes. In the case of nested file scopes, only the innermost nested file scope is searched.

Target Language Functions

The target language function construct is

```
%function identifier ( optional-arguments ) [Output | void]
%return
%endfunction
```

Functions in the target language are recursive and have their own local variable space. Target language functions do not produce output unless they are output functions or explicitly use the `%openfile`, `%selectfile`, and `%closefile` directives.

A function optionally returns a value with the `%return` directive. The returned value can be a type defined in the table at "Target Language Value Types" on page 6-18.

In this example, a function, `name`, returns `x` if `x` and `y` are equal, or returns `z` if `x` and `y` are not equal:

```
%function name(x,y,z) void

%if x == y
   %return x
%else
   %return z
%endif

%endfunction
```

Function calls can appear in contexts where variables are allowed.

The `%with` statements that are in effect when a function is called are available to the function. Calls to other functions do not include the local scope of the function, but do include `%with` statements appearing within the function.

Assignments to variables within a function create new local variables and cannot change the value of global variables unless you use the scope resolution operator (`::`).

By default, a function returns a value and does not produce output. You can override this behavior by specifying the `Output` and `void` modifiers on the function declaration line, as in

```
%function foo() Output
...
%endfunction
```

In this case, the function continues to produce output to the currently open file, and is not required to return a value. You can use the `void` modifier to indicate that the function does not return a value and should not produce output, as in

```
%function foo() void
...
%endfunction
```

Variable Scoping Within Functions

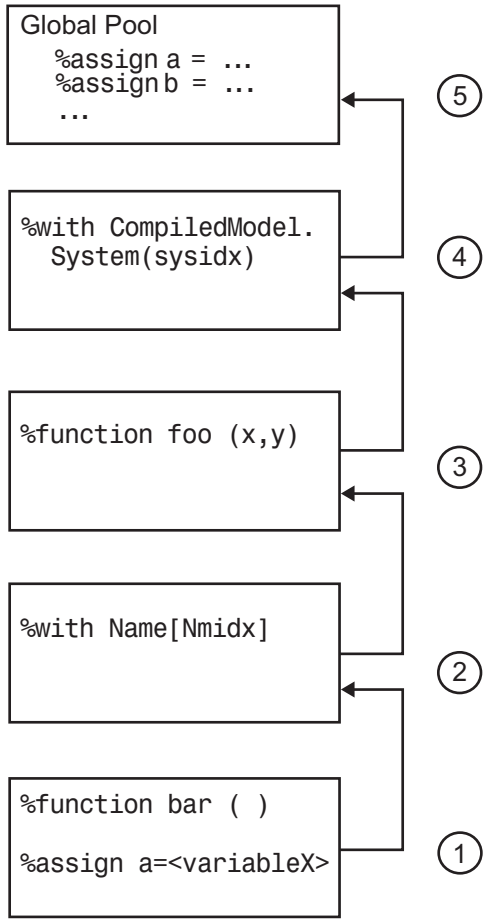
Within a function, the left-hand member of an `%assign` statement defaults to creating a local variable. A new entry is created in the function's block within the scope chain;

it does not affect the other entries. An example appears in “Function Scope” on page 6-58.

You can override this default behavior by using `%assign` with the scope resolution operator (`::`).

When you introduce new scopes within a function, using `%with`, these new scopes are used during nested function calls, but the local scope for the function is not searched.

If a `%with` is included within a function, its associated scope is carried with nested function calls, as shown in the next figure.



%return

The `%return` statement closes all `%with` statements appearing within the current function. In this example, the `%with` statement is automatically closed when the `%return` statement is encountered, removing the scope from the list of searched scopes:

```
%function foo(s)
    %with s
        %return(name)
    %endwith
%endfunction
```

The `%return` statement does not require a value. You can use `%return` to return from a function without a return value.

Command-Line Arguments

In this section...

Target Language Compiler Switches

To call the Target Language Compiler, use

```
tlc [switch1 expr1 switch2 expr2 ...] filename.tlc
```

This table lists the switches you can use with the Target Language Compiler. Order does not makes a difference. Note that if you specify a switch more than once, the last one takes precedence.

Target Language Compiler Switches

Switch	Meaning
`-r` *`filename`*	Reads a database file (such as an `.rtw` file). Repeat this option multiple times to load multiple database files into the Target Language Compiler. Omit this option for target language programs that do not depend on the database.
`-v[`*`number`*`]`	Sets the internal verbosity level to *`number`*. Omitting this option sets the verbosity level to `1`.
`-I`*`path`*	Adds the specified folder to the list of paths to be searched for TLC files.
`-O`*`path`*	Specifies that the output produced should be placed in the designated folder, including files opened with `%openfile` and `%closefile`, and `.log` files created in debug mode. To place files in the current folder, use `-O` (use the capital letter O, not zero).
`-m[`*`number`*`]`	The *`number`* specifies the maximum number of errors to report . Without `-m`, the default is to report the first five errors. If the *`number`* argument is omitted on this option, 1 is assumed.
`-x0`	Parse TLC file only (do not execute).

Switch	Meaning
`-lint`	Performs some simple checks for performance and obsolete features.
`-p[`*`number`*`]`	Prints a dot (`.`) indicating progress for every *`number`* of TLC primitive operations executed.
`-d[a|c|f|n|o]`	Invokes the TLC's debug mode. `-da` makes TLC execute `%assert` directives. However, when using the Simulink Coder build process, this flag is ignored, because it is superseded by the **Enable TLC assertion** check box in the **TLC process** section of the **Code Generation** > **Debug** pane. `-dc` invokes the TLC command-line debugger. `-df` *`filename`* invokes the TLC debugger and runs the debugger script specified by *`filename`*. A debugger script is a text file containing valid debugger commands. TLC searches only the current working folder for the script file. `-dn` causes TLC to produce log files indicating which lines were and were not reached during compilation. `-do` disables the TLC debugging behavior.
`-dr`	Checks for cyclic records (records that reference each other, a source of memory leaks).
`-a[`*`ident`*`]=`*`expr`*	Specifies an initial value, *`expr`*, for the identifier, *`ident`*, for some parameters; equivalent to the `%assign` command.
`-shadow[0|1]`	Enables a warning when an identifier-value pair of a record overwrites a local variable. The warning is disabled by default. `-shadow0` disables the warning. `-shadow1` enables the warning.

As an example, the command line

```
tlc -r myModel.rtw -v grt.tlc
```

specifies that `myModel.rtw` should be read and used to process `grt.tlc` in verbose mode.

Filenames and Search Paths

Target files have the `.tlc` extension. By default, block-level files have the same name as the `Type` of the block in which they appear. You can override the search path for target files with your own local versions. The Target Language Compiler finds target files along this path. If you specify additional search paths with the `-I` switch of the `tlc` command or via the `%addincludepath` directive, the search order is:

1. Current folder.
2. Include paths specified in `%addincludepath` directives. The compiler evaluates multiple `%addincludepath` directives from the *bottom up*.
3. Include paths specified at the command line via `-I` . The compiler evaluates multiple `-I` options from *right to left*.

Note: The compiler does *not* search the MATLAB path, and will not find a file that is available only on that path. The compiler searches only the locations described above.

Debugging TLC Files

The Target Language Compiler debugger is a command-line debugger that enables you to identify problems in executing TLC code. The following sections describe the facilities provided and provide examples of use.

About the TLC Debugger

In this section...

TLC Debugger Overview

The TLC debugger helps you identify programming errors in your TLC code. Using the debugger, you can execute TLC code line-by-line, analyze and/or change variables in a specified block scope, and view the TLC call stack. The TLC debugger has a command-line interface that provides commands similar to standard debugging tools such as `dbx` or `gdb`.

Tips for Debugging TLC Code

Here are a few tips that will help you to debug your TLC code:

1. To see the full TLC call stack, place the following statement in your TLC code before the line that is pointed to by the error message. This will be helpful in narrowing down your problem.

   ```
   %setcommandswitch "-v1"
   ```

2. To trace the value of a variable in a function, place the following statement in your TLC file:

   ```
   %trace This is in my function %<variable>
   ```

   Your message will appear when the Target Language Compiler is run with the `-v` command switch, but not otherwise. You can use `%warning` instead of `%trace` to print variables, but you will need to remove or comment out such lines after you are through debugging.

3. Use the TLC coverage log files to identify parts of your code that have not been reached.

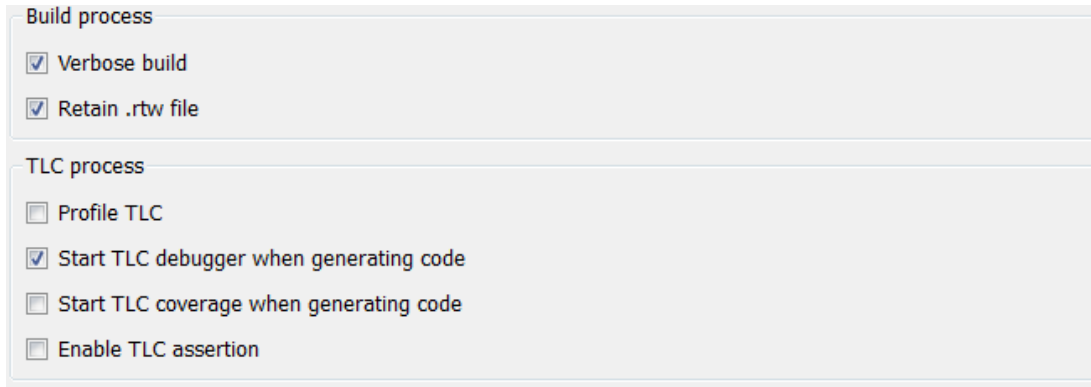
Using the TLC Debugger

In this section...

Invoking the Debugger

Use the TLC debugger to identify bugs and potential problems in your TLC files. To use the TLC debugger:

1. Configure TLC for debugging via the Configuration Parameters dialog, by selecting **Debug** under **Code Generation**.
2. Select **Retain .rtw file**. This prevents the *`model`*`.rtw` file from being deleted after code generation.
3. Select **Start TLC debugger when generating code** to invoke the TLC debugger when starting the code generation process. The **Debug** pane of the dialog box looks like this.



   Selecting **Start TLC debugger when generating code** is equivalent to specifying the TLC option `-dc` on the **Code Generation** pane of the Configuration Parameters dialog box.
4. Apply your changes and click **Build** to start code generation. This stops at the first line of executed TLC code, breaks into the TLC command-line debugger, and displays the following prompt:

```
TLC_DEBUG>
```

You can now set breakpoints, explore the contents of Simulink Coder files, and explore variables in your TLC file using `print`, `which`, or `whos`.

An alternative way to invoke the TLC debugger is from the MATLAB prompt. (This assumes you retained the *`model`*`.rtw` file in the project folder.) To avoid making mistakes, copy the `tlc` command output of the build process to the MATLAB Command Window, and issue it after appending `-dc` to that command line.

A complete list of command-line switches for the TLC debugger is available in the table "Target Language Compiler Switches" on page 6-64.

TLC Debugger Command Summary

The table TLC Debugger Commands summarizes the TLC debugger commands.

To obtain more detailed help on individual commands, use the syntax

```
help command
```

from within the TLC debugger, as in this example:

```
TLC-DEBUG> help clear
```

You can abbreviate a TLC debugger command to its shortest unique form. For example,

```
TLC-DEBUG> break warning
```

can be abbreviated to

```
TLC-DEBUG> br warning
```

To view a complete list of TLC debugger commands, type `help` at the `TLC-DEBUG>` prompt.

TLC Debugger Commands

Command	Description
`assign variable=value`	Change a variable in the running program.
`break ["filename":]line|error|warning|`	Set a breakpoint. See also "%breakpoint Directive" on page 7-6.

Command	Description
`trace|function`	
`clear [breakpoint#|all]`	Remove a breakpoint.
`condition [breakpoint#] [expression]`	Attach a condition to a breakpoint.
`continue ["filename":]line|function`	Continue from a breakpoint.
`disable [breakpoint#]`	Disable a breakpoint.
`down [n]`	Move down the stack.
`enable [breakpoint#]`	Enable a breakpoint.
`finish`	Break after completing the current function.
`help [command]`	Obtain help for a command.
`ignore [breakpoint#]count`	Set the ignore count of a breakpoint.
`iostack`	Display contents of I/O stack.
`list start[,end]`	List lines from the file from start to end.
`loadstate "filename"`	Load debugger breakpoint state from a file.
`next`	Single step without going into functions.
`print expression`	Print the value of a TLC expression. To print a record, you must specify a fully qualified scope such as `CompiledModel.System[0].Block[0]`.
`quit`	Quit the TLC debugger. You can also exit the debugger by typing **Ctrl+C** at the prompt.
`run "filename"`	Run a batch file of command-line debugger commands.
`savestate "filename"`	Save debugger breakpoint state to a file.
`status`	Display a list of active breakpoints.
`step`	Step into.
`stop ["filename":]line|error|warning|trace|function`	Set a breakpoint (same as `break`).
`tbreak ["filename":]line|function`	Set a temporary breakpoint.
`thread [n]`	Change the active thread to thread `#n` (0 is the main program's thread number).

Command	Description
`threads`	List the currently active TLC execution threads.
`tstop ["filename":]line|function`	Set a temporary breakpoint.
`up [n]`	Move up the stack.
`where`	Show the currently active execution chains.
`which name`	Look up the name and display what scope it comes from.
`whos [::|expression]`	List the variables in the given scope.

%breakpoint Directive

As an alternative to the `break` command, you can embed breakpoints at locations in a TLC file by adding the directive

```
%breakpoint
```

Usage Notes

When using `break` or `stop`, use

- `error` to break or stop on error
- `warn` to break or stop on warning
- `trace` to break or stop on trace

For example, if you need to break on error, use

```
TLC_DEBUG> break error
```

When using `clear`, get the status of breakpoints using `status` and clear specific breakpoints. For example

```
TLC-DEBUG> break "foo.tlc":46
TLC-DEBUG> break "foo.tlc":25
TLC-DEBUG> status
Breakpoints:
[1] break File: foo.tlc Line: 46
[2] break File: foo.tlc Line: 25
TLC-DEBUG> clear 2
```

In this example, `clear 2` clears the second breakpoint.

TLC Coverage

In this section...

Using the TLC Coverage Option

The example in the last section used the debugger to detect a problem in one section of the TLC file. Because a test model may not cover all possible cases, there is a technique that traces the untested cases, the TLC coverage option.

The TLC coverage option provides an easier way to ascertain that the different code parts (not paths) in your code are exercised. To specify TLC coverage tracking, select **Start TLC coverage when generating code** on the **Code Generation > Debug** pane of the Configuration Parameters dialog box:

When you initiate TLC coverage, the Target Language Compiler produces a `.log` file for every target file (`*.tlc`) used. These `.log` files are placed in project folder created for the model. Each `.log` file contains usage (count) information regarding how many times it encounters each line during execution. Each line begins with the number of times it is encountered, followed by a colon, followed by the code.

Example .log File

Here is a log file that results from generating code for the example model `sfcndemo_sdotproduct`, located in

`matlabroot``/toolbox/simulink/simdemos/simfeatures`
This model inlines the `sdotproduct` S-function in TLC. The TLC file that implements the S-function is located in *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/tlc_c/`. The `.log` file for `sdotproduct.tlc` is `sdotproduct.log`, which is placed in your build folder. The contents of `sdotproduct.log` are similar to:

```
Source: E:\matlab\toolbox\simulink\simdemos\simfeatures\tlc_c\sdotproduct.tlc
 0: %%
 0: %% File : sdotproduct.tlc generated from sdotproduct.ttlc revision 1.6
 0: %%
 0: %% Abstract:
 0: %%      Dot product block target file.
 1:
 1: %implements sdotproduct "C"
 1:
 0: %% Function: FcnThriftedComplexMultiply
========================================
 0: %% Abstract:
 0: %%      This function multiplies two numbers in the complex plane. If any of
 0: %%      the input arguments is only real, then the complex part is passed in
 0: %%      as "".
 0: %%
 1: %function FcnThriftedComplexConjMultiply(ar,ai,br,bi,cr,ci,op) void
 2:   %openfile buffer
 0:   %%
 0:   %% Compute Cr = Ar * Br + Ai * Bi
 0:   %%
 2:   %assign rhsStr = "%<ar> * %<br>"
 2:   %if !LibIsEqual(ai, "") && !LibIsEqual(bi, "")
 0:     %assign rhsStr = rhsStr + " + %<ai> * %<bi>"
 0:   %endif
 2:   %<cr> %<op> %<rhsStr>;
 0:   %%
 0:   %% Compute Ci = Ar * Bi - Ai * Br
 0:   %%
 2:   %if !LibIsEqual(ci, "")
 0:     %assign rhsStr = "0.0"
 0:     %if !LibIsEqual(bi, "")
 0:       %assign rhsStr = "%<ar> * %<bi>"
 0:     %endif
 0:     %if !LibIsEqual(ai, "")
 0:       %assign rhsStr = rhsStr + " - %<ai> * %<br>"
 0:     %endif
 0:     %<ci> %<op> %<rhsStr>;
 0:   %endif
 0:   %%
 2:   %closefile buffer
```

```
 2:   %return buffer
 0: %endfunction %% FcnThriftedComplexMultiply
 1:
 1:
 0: %% Function: Outputs
============================================================
 0: %% Abstract:
 0: %%      Y = U0' * U1, where U0' is the complex conjugate transpose of U0
 0: %%
 1: %function Outputs(block, system) Output
 1:   %assign sfcnName = ParamSettings.FunctionName
 1:   /* %<Type> Block (%<sfcnName>): %<LibParentMaskBlockName(block)> */
 0:   %%
 1:   %assign u0re = LibBlockInputSignal(0, "", "", "%<tRealPart>0")
 1:   %assign u0im = LibBlockInputSignal(0, "", "", "%<tImagPart>0")
 1:   %assign u1re = LibBlockInputSignal(1, "", "", "%<tRealPart>0")
 1:   %assign u1im = LibBlockInputSignal(1, "", "", "%<tImagPart>0")
 0:   %%
 1:   %assign yre = LibBlockOutputSignal(0, "", "", "%<tRealPart>0")
 1:   %assign yim = LibBlockOutputSignal(0, "", "", "%<tImagPart>0")
 0:   %%
 0:   %% Need to declare a temporary variable for u1re when the output is
 0:   %% being overwritten and u0im is nonzero
 1:   %assign outputOverWritesInput = ...
 0:     ((LibBlockInputSignalBufferDstPort(0) == 0) || ...
 0:      (LibBlockInputSignalBufferDstPort(1) == 0)) &&   ...
 0:     (LibBlockInputSignalIsComplex(0) && LibBlockInputSignalIsComplex(1))
 0:   %%
 1:   %if outputOverWritesInput
 0:     {
 0:       %assign dtName = LibBlockOutputSignalDataTypeName(0, tRealPart)
 0:       %<dtName> tmpVar;
 0:     \
 0:     %assign tmpVar = "tmpVar"
 0:   %else
 1:     %assign tmpVar = yre
 0:   %endif
 0:   %%
 1:   %<FcnThriftedComplexConjMultiply(u0re, u0im, u1re, u1im, tmpVar, yim, "=")>\
 0:   %%
 1:   %assign rollVars    = ["U", "Y"]
 1:   %assign rollRegion  = LibGetRollRegions1(RollRegions)
 0:   %%
 1:   %if LibIsEqual(rollRegion, [])
 0:     %if outputOverWritesInput
 0:       %<yre> = tmpVar;
 0:     %endif
 0:   %else
 0:     %% Continue with dot product for nonscalar case
 1:     %roll idx = rollRegion, lcv = RollThreshold, block, "Roller", rollVars
 1:       %assign u0re = LibBlockInputSignal(0,"",lcv,"%<tRealPart>%<idx>")
 1:       %assign u0im = LibBlockInputSignal(0,"",lcv,"%<tImagPart>%<idx>")
 1:       %assign u1re = LibBlockInputSignal(1,"",lcv,"%<tRealPart>%<idx>")
 1:       %assign u1im = LibBlockInputSignal(1,"",lcv,"%<tImagPart>%<idx>")
 0:       %%
```

```
1:       %assign yre = LibBlockOutputSignal(0,"",lcv,"%<tRealPart>%<idx>")
1:       %assign yim = LibBlockOutputSignal(0,"",lcv,"%<tImagPart>%<idx>")
0:       %%
1:       %<FcnThriftedComplexConjMultiply(uOre, uOim, u1re, u1im, yre, yim, "+=")>\
0:     %endroll
0:   %endif
1:   %if outputOverWritesInput
0:     }
0:   %endif
1:
0: %endfunction
1:
0: %% [EOF] sdotproduct.tlc
```

Analyzing the Results

This structure makes it easy to identify branches not taken and to develop new tests that can exercise unused portions of the target files.

Looking at the `sdotproduct.log` file, you can see that the code has not been used to assign default values to parameters (e.g., the first part of the code for function `FcnThriftedComplexConjMultiply`). Using this log as a reference and creating models that exercise unexecuted lines, you can make sure that your code is more robust.

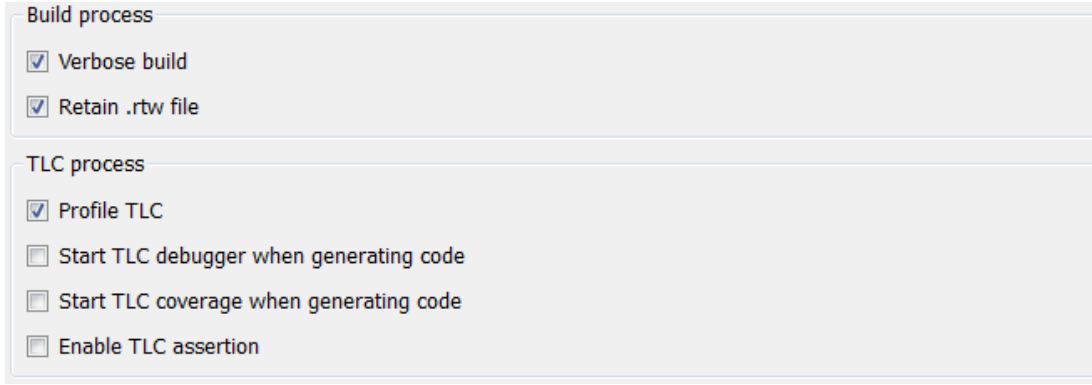
TLC Profiler

In this section...

Using the Profiler

The TLC profiler collects timing statistics for TLC code. It collects execution time for functions, scripts, macros, and built-in functions. These results become the basis of HTML reports that are identical in format to MATLAB profiler reports. By analyzing the report, you can identify bottlenecks in your code that make code generation take longer.

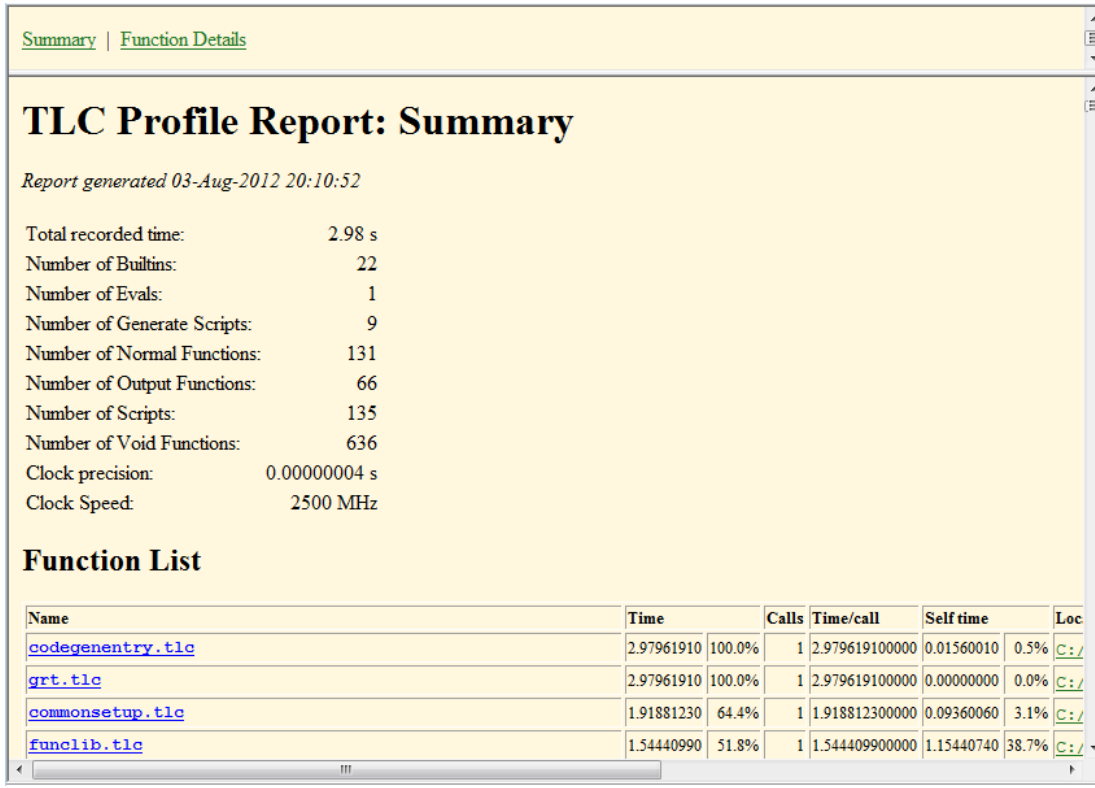
To access the profiler, select **Profile TLC** on the **Code Generation > Debug** pane of the Configuration Parameters dialog box. Apply your changes and click the **Build** (or **Generate code**) button.



At the end of the TLC process, the build process creates the HTML summary and related files.

Analyzing the Report

The profile report is generated into the Simulink Coder build folder. To open the report, change folder (`cd`) to the build folder and open the file *`model`*`.html`, opening it in a browser window. Here is a sample of a TLC profiling report:



Summary | Function Details

TLC Profile Report: Summary

Report generated 03-Aug-2012 20:10:52

Total recorded time:	2.98 s
Number of Builtins:	22
Number of Evals:	1
Number of Generate Scripts:	9
Number of Normal Functions:	131
Number of Output Functions:	66
Number of Scripts:	135
Number of Void Functions:	636
Clock precision:	0.00000004 s
Clock Speed:	2500 MHz

Function List

Name	Time		Calls	Time/call	Self time		Loc
codegenentry.tlc	2.97961910	100.0%	1	2.979619100000	0.01560010	0.5%	C:/
grt.tlc	2.97961910	100.0%	1	2.979619100000	0.00000000	0.0%	C:/
commonsetup.tlc	1.91881230	64.4%	1	1.918812300000	0.09360060	3.1%	C:/
funclib.tlc	1.54440990	51.8%	1	1.544409900000	1.15440740	38.7%	C:/

The created report is fairly self-explanatory. Some points to note are

- Functions are sorted in descending order of their execution time.
- Self-time is the time spent in the function alone, not including the time spent in functions called by the function.
- Functions are hyperlinks that take you to the details related to that specific function.

The profiler report can be helpful when you have inlined S-functions in your model. You can use the profiler to compare time spent in specific user-written or `Lib` functions, and then modify your TLC code accordingly.

Nonexecutable Directives

TLC considers the following directives to be nonexecutable lines. Therefore, these directives are not counted in TLC Profiler reports:

- `%filescope`
- `%else`
- `%endif`
- `%endforeach`
- `%endfor`
- `%endroll`
- `%endwith`
- `%body`
- `%endbody`
- `%endfunction`
- `%endswitch`
- `%default`
- Comment (`%%` or `/% text %/`

Improving Performance

Analyzing the profiler results also gives you an overview of which functions are used more often or are more expensive. Then, you can either improve those functions, or try alternative methods to improve code generation speed. Two points to consider are

- Reduce usage of `EXISTS`. Performing an `EXISTS` on a field is more costly than comparing the field to a value. When possible, create an inert default value for a field. Then, instead of doing an `EXISTS` on the entity, compare it against the default value.
- Reduce the use of one-line functions. One-line functions might be a bottleneck for code generation speed. When readability is not an issue, consider expanding the function.

Inlining S-Functions

To wrap or to inline, that is the question. Once you have decided, the following sections explain how to go about it, using the `timestwo` S-function as a running example. Inlining works almost identically for C, C++, MATLAB file, and Fortran S-functions.

Inline S-Functions in Generated Code

Writing S-functions to be included in generated code involves requirements that go beyond writing S-functions used only for simulation. Before you proceed to inline an S-function make sure that it meets requirements and functions as you expect. For more information, see “S-Functions and Code Generation” in the Simulink Coder documentation. If your S-function is multirate, see “Time-Based Scheduling and Code Generation” and “Modeling for Multitasking Execution” in the Simulink Coder documentation, and “Rate Grouping Compliance and Compatibility Issues” in the Embedded Coder documentation.

Inline S-Functions with Block Target Files

In this section...

When to Inline S-Functions

With C MEX S-functions, non-ERT targets support calling the original C MEX code if the source code (`.c` file) is available when entering the build phase. For S-functions that are in Fortran or MATLAB language, you must inline them to have complete code generation for Simulink models that contain them. Additionally, once you have determined that you will inline an S-function, you must decide to make it either fully inlined or wrapped.

Fully Inlined S-Functions

The block target file for a fully inlined S-function is a self-contained definition of how to inline the block's functionality directly into the various portions of the generated code — start code, output code, etc. This approach is most beneficial when there are many modes and data types supported for algorithms that are relatively small or when the code size is not significant.

Function-Based or Wrapped Code Generation

When the physical size of the code for a block becomes too large for inlining, the block target file is written to gather inputs, outputs, and parameters, and make a call to a function that you write to perform the block functionality. This has an advantage in generated code size when the code in the function is large or there are many instances of this block in a model. Of course, you should consider the overhead of the function call when weighing the option of fully inlining the block algorithm or generating function calls.

If you choose to go with function-based code generation, two more options need consideration:

- Write the functions once, put them in `.c` files, and have the TLC code's `BlockTypeSetup` method specify external references to your support functions.

Use `LibAddToModelSources` for names of the modules containing the supporting functions. This approach is usually done using one function per file to get the smallest executable possible.

- Write a more sophisticated TLC file.In addition to methods such as `Start` and `Outputs`, conditionally generate customized versions of functions (data types, widths, algorithms, and so on), in separate code generation buffers, to be written to a separate `.c` file. The file should contain only functions used by this model, instead of all possible functions.

Either approach can produce optimal code. The first option can result in hundreds of files if your S-function supports many data types, signal widths, and algorithm choices. The second approach is more difficult to write, but results in a more maintainable code generation library, and the code can be every bit as tight as the first approach.

For further information on wrapping, see “Wrapper Inlined S-Function Example” on page 2-11.

Inline C MEX S-Functions

In this section...

Inline S-Function Overview

When a Simulink model contains an S-function and a corresponding TLC block target file exists for that S-function, the code generator inlines the S-function. Inlining an S-function can produce more efficient code by eliminating the S-function API layer from the generated code.

For S-functions that can perform a variety of tasks, inlining them gives you the opportunity to generate code only for the current mode of operation set for each instance of the block. As an example of this, if an S-function accepts an arbitrary signal width and loops through each element of the signal, you would want to generate inlined code that has loops when the signal has two or more elements, but generates a simple nonlooped calculation when the signal has just one element.

Level 1 C MEX S-functions (written to an older form of the S-function API) that are not inlined will cause the generated code to make calls to all of these functions even if the routine is empty for the particular S-function.

Function	Purpose
`mdlInitializeSizes`	Initialize the sizes array
`mdlInitializeSampleTimes`	Initialize the sample times array
`mdlInitializeConditions`	Initialize the states
`mdlOutputs`	Compute the outputs
`mdlUpdate`	Update discrete states
`mdlDerivatives`	Compute the derivatives of continuous states
`mdlTerminate`	Clean up when the simulation terminates

Level 2 C MEX S-functions (i.e., those written to the current S-function API) that are not inlined make calls to the above functions, with the following exceptions:

- `mdlInitializeConditions` is called only if `MDL_INITIALIZE_CONDITIONS` is declared with `#define`.
- `mdlStart` is called only if `MDL_START` is declared with `#define`.
- `mdlUpdate` is called only if `MDL_UPDATE` is declared with `#define`.
- `mdlDerivatives` is called only if `MDL_DERIVATIVES` is declared with `#define`.

By inlining an S-function, you can eliminate the calls to these possibly empty functions in the simulation loop. This can greatly improve the efficiency of the generated code.

To inline an S-function called *`sfunc`*`_name`, you create a custom S-function block target file called *`sfunc`*`_name.tlc` and place it in the same folder as the S-function MEX-file. Then, at build time, the target file is executed instead of setting up function calls into the S-function `.c` file. The S-function target file "inlines" the S-function by directing the Target Language Compiler to insert only the statements defined in the target file.

In general, inlining an S-function is especially useful when

- The time required to execute the contents of the S-function is small in comparison to the overhead required to call the S-function.
- Certain S-function routines are empty (e.g., `mdlUpdate`).
- The behavior of the S-function changes between simulation and code generation. For example, device driver I/O S-functions might read from the MATLAB workspace during simulation, but read from an actual hardware address in the generated code.

S-Function Parameters

An S-function can write two different types of parameters into the *model*.rtw file for Target Language Compiler files to access:

- Parameter settings: These correspond to nontunable parameters (typically set from check boxes and menus on a masked S-function) that are written via the `mdlRTW` method of the S-function using `ssWriteRTWParamSettings`. The S-function TLC implementation file can then directly access the values of these parameter settings from the `SFcnParamSettings` record in the block.
- Tunable parameters: This class of parameters can be accessed when they are registered as run-time parameters within the S-function. Note that such tunable parameters are automatically written out to the *model*.rtw file. Within the TLC file for the S-function, you can access run-time parameters and their attributes using the `LibBlockParameter` library function and its variants.

See "Run-Time Parameters" in the Simulink Writing S-Functions documentation for more information on how to create and use run-time parameters. Also see the example `sfcndemo_runtime` in the S-function examples for how to create and use the two classes of parameters. The example source files, which you can inspect and adapt, are

- `toolbox/simulink/simdemos/simfeatures/src/sfun_runtime1.c`
- `toolbox/simulink/simdemos/simfeatures/tlc_c/sfun_runtime1.tlc`
- `toolbox/simulink/simdemos/simfeatures/src/sfun_runtime2.c`
- `toolbox/simulink/simdemos/simfeatures/tlc_c/sfun_runtime2.tlc`
- `toolbox/simulink/simdemos/simfeatures/src/sfun_runtime3.c`
- `toolbox/simulink/simdemos/simfeatures/tlc_c/sfun_runtime3.tlc`

Sample Code for S-Function

Suppose you have a simple S-function that mimics the Gain block, with one input, one output, and a scalar gain. That is, `y = u * p`. If the Simulink block's name is `foo` and the name of the Level 2 S-function is `foogain`, the C MEX S-function must contain this code:

```
#define S_FUNCTION_NAME foogain
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"
#define GAIN mxGetPr(ssGetSFcnParam(S,0))[0]

static void mdlInitializeSizes(SimStruct *S)
{
  ssSetNumContStates  (S, 0);
  ssSetNumDiscStates  (S, 0);

  if (!ssSetNumInputPorts(S, 1)) return;
  ssSetInputPortWidth            (S, 0, 1);
  ssSetInputPortDirectFeedThrough(S, 0, 1);

  if (!ssSetNumOutputPorts(S, 1)) return;
  ssSetOutputPortWidth          (S, 0, 1);

  ssSetNumSFcnParams  (S, 1);
  ssSetNumSampleTimes  (S, 0);
  ssSetNumIWork  (S, 0);
  ssSetNumRWork  (S, 0);
  ssSetNumPWork  (S, 0);
}

static void
mdlOutputs(SimStruct *S, int_T tid)
{
  real_T *y = ssGetOutputPortRealSignal(S, 0);
  const InputRealPtrsType u = ssGetInputPortRealSignalPtrs(S, 0);

  y[0] = (*u)[0] * GAIN;
}
```

```
static void
mdlInitializeSampleTimes(SimStruct *S){}

static void
mdlTerminate(SimStruct *S) {}

#define MDL_RTW  /* Change to #undef to remove function */
#if defined(MDL_RTW)&&(defined(MATLAB_MEX_FILE)||defined(NRT))
static void
mdlRTW (SimStruct *S)
{
  if (!ssWriteRTWParameters(S, 1,SSWRITE_VALUE_VECT,"Gain","",
                            mxGetPr(ssGetSFcnParam(S,0)),1))
  {
    return;
  }
}
#endif

#ifdef  MATLAB_MEX_FILE
#include "simulink.c"
#else
#include "cg_sfun.h"
#endif
```

The following two sections show the difference in the generated code for *`model`*`.c` containing noninlined and inlined versions of S-function `foogain`. The model contains no other Simulink blocks.

For information about how to generate code, see “Initiate Code Generation” and “Program Builds”.

Comparison of Noninlined and Inlined Versions of model.c

Without a TLC file to define the S-function specifics, the Simulink Coder code generator must call the MEX-file S-function through the S-function API. The following code is the *`model`*`.c` file for the noninlined S-function (i.e., no corresponding TLC file exists).

Noninlined S-Function

```
/*
 * model.c
.
.
```

```
.
*/
real_T untitled_RGND = 0.0;             /* real_T ground */
/* Start the model */
void MdlStart(void)
{
  /* (no start code required) */
}
/* Compute block outputs */
void MdlOutputs(int_T tid)
{
  /* Level2 S-Function Block: <Root>/S-Function (foogain) */
  {
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnOutputs(rts, tid);
  }
}
/* Perform model update */
void MdlUpdate(int_T tid)
{
  /* (no update code required) */
}
/* Terminate function */
void MdlTerminate(void)
{
  /* Level2 S-Function Block: <Root>/S-Function (foogain) */
  {
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnTerminate(rts);
  }
}
#include "model_reg.h"
/* [EOF] model.c */
```

Inlined S-Function

This code is *model*.c with the `foogain` S-function fully inlined:

```
/*
 * model.c
.
.
.
*/
/* Start the model */
```

```
void MdlStart(void)
{
  /* (no start code required) */
}

/* Compute block outputs */
void MdlOutputs(int_T tid)

  /* S-Function block: <Root>/S-Function */
  /* NOTE: There are no calls to the S-function API in the inlined
     version of model.c. */
  rtB.S_Function = 0.0 * rtP.S_Function_Gain;
}

/* Perform model update */
void MdlUpdate(int_T tid)
{
  /* (no update code required) */
}

/* Terminate function */
void MdlTerminate(void)
{
  /* (no terminate code required) */
}

#include "model_reg.h"

/* [EOF] model.c */
```

If you include this target file for this S-function block, the resulting *`model`*`.c` code is

```
rtB.S_Function = 0.0 * rtP.S_Function_Gain;
```

Including a TLC file drastically decreased the code size and increased the execution efficiency of the generated code. These notes highlight some information about the TLC code and the generated output:

- The TLC directive `%implements` is required by block target files, and must be the first executable statement in the block target file. This directive prevents the Target Language Compiler from executing an inappropriate target file for S-function `foogain`.
- The input to `foo` is `rtGROUND` (a Simulink Coder global equal to 0.0) because `foo` is the only block in the model and its input is unconnected.

- Including a TLC file for `foogain` eliminates the need for an S-function registration segment for `foogain`. This significantly reduces code size.
- The TLC code inlines the `gain` parameter when the Simulink Coder build process is configured to inline parameter values. For example, if the S-function parameter is specified as 2.5 in the S-function dialog box, the TLC `Outputs` function generates

  ```
  rtB.foo = input * 2.5;
  ```
- Use the `%generatefile` directive if your operating system has a filename size restriction and the name of the S-function is `foosfunction` (that exceeds the limit). In this case, you would include the following statement in the system target file (anywhere prior to a reference to this S-function block target file).

  ```
  %generatefile foosfunction "foosfunc.tlc"
  ```

  This statement tells the Target Language Compiler to open `foosfunc.tlc` instead of `foosfunction.tlc`.

Comparison of Noninlined and Inlined Versions of model_reg.h

Inlining a Level 2 S-function significantly reduces the size of the *`model`*`_reg.h` code. Model registration functions are lengthy; much of the code has been eliminated in this example. The code below highlights the difference between the noninlined and inlined versions of *`model`*`_reg.h`; inlining eliminates this code:

```
/*
 * model_reg.h
 *
*/
/* Normal model initialization code independent of
      S-functions  */

/* child S-Function registration */
  ssSetNumSFunctions(rtS, 1);

  /* register each child */
  {
    static SimStruct childSFunctions[1];
    static SimStruct *childSFunctionPtrs[1];

    (void)memset((char_T *)&childSFunctions[0], 0,
                 sizeof(childSFunctions));
    ssSetSFunctions(rtS, &childSFunctionPtrs[0]);
    {
```

```
  int_T i;

  for(i = 0; i < 1; i++) {
    ssSetSFunction(rtS, i, &childSFunctions[i]);
  }
}

/* Level2 S-Function Block: untitled/<Root>/S-Function
   (foogain) */
{
  extern void foogain(SimStruct *rts);
  SimStruct *rts = ssGetSFunction(rtS, 0);

  /* timing info */
  static time_T sfcnPeriod[1];
  static time_T sfcnOffset[1];
  static int_T sfcnTsMap[1];

  {
    int_T i;

    for(i = 0; i < 1; i++) {
      sfcnPeriod[i] = sfcnOffset[i] = 0.0;
    }
  }
  ssSetSampleTimePtr(rts, &sfcnPeriod[0]);
  ssSetOffsetTimePtr(rts, &sfcnOffset[0]);
  ssSetSampleTimeTaskIDPtr(rts, sfcnTsMap);
  ssSetMdlInfoPtr(rts, ssGetMdlInfoPtr(rtS));

  /* inputs */
  {
    static struct _ssPortInputs inputPortInfo[1];

    _ssSetNumInputPorts(rts, 1);
    ssSetPortInfoForInputs(rts, &inputPortInfo[0]);

    /* port 0 */
    {
      static real_T const *sfcnUPtrs[1];

      sfcnUPtrs[0] = &untitled_RGND;
      ssSetInputPortWidth(rts, 0, 1);
      ssSetInputPortSignalPtrs(rts, 0,
```

```
        (InputPtrsType)&sfcnUPtrs[0]);
  }
}

/* outputs */
{
  static struct _ssPortOutputs outputPortInfo[1];
  _ssSetNumOutputPorts(rts, 1);
  ssSetPortInfoForOutputs(rts, &outputPortInfo[0]);
  ssSetOutputPortWidth(rts, 0, 1);
  ssSetOutputPortSignal(rts, 0, &rtB.S_Function);
}

/* path info */
ssSetModelName(rts, "S-Function");
ssSetPath(rts, "untitled/S-Function");
ssSetParentSS(rts, rtS);
ssSetRootSS(rts, ssGetRootSS(rtS));
ssSetVersion(rts, SIMSTRUCT_VERSION_LEVEL2);

/* parameters */
{
  static mxArray const *sfcnParams[1];

  ssSetSFcnParamsCount(rts, 1);
  ssSetSFcnParamsPtr(rts, &sfcnParams[0]);

  ssSetSFcnParam(rts, 0, &rtP.S_Function_P1Size[0]);
}

/* registration */
foogain(rts);

sfcnInitializeSizes(rts);
sfcnInitializeSampleTimes(rts);

/* adjust sample time */
ssSetSampleTime(rts, 0, 0.2);
ssSetOffsetTime(rts, 0, 0.0);
sfcnTsMap[0] = 0;

/* Update the InputPortReusable and BufferDstPort flags for
  each input port */
ssSetInputPortReusable(rts, 0, 0);
```

```
      ssSetInputPortBufferDstPort(rts, 0, -1);

      /* Update the OutputPortReusable flag of each output port */
    }
  }
```

A TLC File to Inline S-Function foogain

To avoid unnecessary calls to the S-function and to generate the minimum code required for the S-function, the following TLC file, `foogain.tlc`, is provided as an example.

```
%implements "foogain" "C"

%function Outputs (block, system) Output
  /* %<Type> block: %<Name> */
  %%
  %assign y = LibBlockOutputSignal (0, "", "", 0)
  %assign u = LibBlockInputSignal (0, "", "", 0)
  %assign p = LibBlockParameter (Gain, "", "", 0)
  %<y> = %<u> * %<p>;
%endfunction
```

Managing Block Instance Data with an Eye Toward Code Generation

Instance data is extra data or working memory that is unique to each instance of a block in a Simulink model. This does not include parameter or state data (which is stored in the model parameter and state vectors, respectively), but rather is used to cache intermediate results or derived representations of parameters and modes. One example of instance data is the buffer used by a transport delay block.

Allocating and using memory on an instance-by-instance basis can be done several ways in a Level 2 S-function: via `ssSetUserData`, work vectors (e.g., `ssSetRWork`, `ssSetIWork`), or data-typed work vectors known as `DWork` vectors. For the smallest effort in writing the S-function and block target file and for automatic conformance to both static and `malloc` instance data on targets such as `grt`, use data-typed work vectors when writing S-functions with instance data.

The advantages are twofold. In the first place, writing the S-function is more straightforward, in that memory allocations and frees are handled for you by Simulink. Secondly, the `DWork` vectors are written to the *`model`*`.rtw` file for you automatically, including the `DWork` name, data type, and size. This makes writing the block target file easier, because you do not have to write TLC code for allocating and freeing the `DWork` memory.

Additionally, if you want to bundle groups of `DWork` vectors into structures for passing to functions, you can populate the structure with pointers to `DWork` arrays in both your S-function `mdlStart` function and the block target file's `Start` method, achieving consistency between the S-function and the generated code's handling of data.

Finally, using a `DWork` makes it straightforward to create a specific version of code (data types, scalar vs. vectorized, etc.) for each block instance that matches the implementation in the S-function. Both implementations use `DWork` in the same way so that the inlined code can be used with the Simulink Accelerator™ software without changes to the C MEX S-function or the block target file.

Using Inlined Code with the Simulink Accelerator Software

By default, the Simulink Accelerator software calls your C MEX S-function as part of an accelerated model simulation. If you prefer to have the accelerator inline your S-function before running the accelerated model, tell the accelerator to use your block target file to inline the S-function with the `SS_OPTION_USE_TLC_WITH_ACCELERATOR` flag in the call to `ssSetOptions()` in the `mdlInitializeSizes` function of that S-function.

Note that memory and work vector size and usage must be the same for the TLC generated code and the C MEX S-function, or the Simulink Accelerator software cannot execute the inlined code properly. This is because the C MEX S-function is called to initialize the block and its work vectors, calling the `mdlInitializeSizes`, `mdlInitializeConditions`, `mdlCheckParameters`, `mdlProcessParameters`, and `mdlStart` functions. In the case of constant signal propagation, `mdlOutputs` is called from the C MEX S-function during the initialization phase of model execution.

During the time-stepping phase of accelerated model execution, the code generated by the `Output` and `Update` block TLC methods will execute, plus the `Derivatives` and zero-crossing methods if they exist. The `Start` method of the block target file is not used in generating code for an accelerated model.

Inline MATLAB File S-Functions

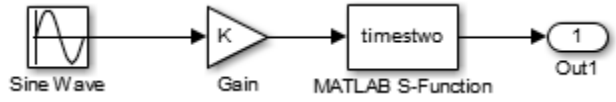
The functionality of MATLAB file S-functions can be inlined in the generated code. Writing a block target file for a MATLAB file S-function is essentially identical to the process for a C MEX S-function.

Note: While you can fully inline a MATLAB file S-function to improve performance, a C or C++ API for the MATLAB Math Library is not included with Simulink Accelerator or the Simulink Coder software. You therefore cannot call MATLAB Math Library functions from a TLC file.

The following example illustrates the equivalence of C MEX and MATLAB file S-functions for code generation. The S-function MATLAB file `timestwo.m` is equivalent to the C MEX S-function `timestwo`. In fact, the TLC file for the C MEX S-function `timestwo` works for the S-function MATLAB file `timestwo.m` as well. Because TLC requires only the root name of the S-function and not its type, it is independent of the type of S-function. In the case of `timestwo`, one line determines how the TLC file will be used:

```
%implements "timestwo" "C"
```

To try this yourself, copy file `timestwo.m` from *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/` to a temporary folder, then copy the file `timestwo.tlc` from *`matlabroot`*`/toolbox/simulink/simdemos/simfeatures/tlc_c/` to the same temporary folder. In MATLAB, change folder (`cd`) to the temporary folder and make a Simulink model with an S-function block that calls `timestwo`. Here is the sample model:



Because the MATLAB search path will find `timestwo.m` in the current folder before finding the C MEX S-function `timestwo` in the `matlabpath`, Simulink uses the MATLAB file S-function for simulation. Verify which S-function will be used by typing the MATLAB command

```
which timestwo
```

The answer you see will be the MATLAB file S-function `timestwo.m` in the temporary folder.

Upon generating code, you will find that the `timestwo.tlc` file was used to inline the MATLAB file S-function with code that looks like this (with an input signal width of 5 in this example):

```
/* S-Function Block: <Root>/MATLAB S-Function */
  /* Multiply input by two */
  {
    int_T i1;
    const real_T *u0 = &rtB.Gain[0];
    real_T *y0 = &rtB.m_file_S_Function[0];

    for (i1=0; i1 < 5; i1++) {
      y0[i1] = u0[i1] * 2.0;
    }
  }
```

As expected, each of the inputs, `u0[i1]`, is multiplied by 2.0 to form the output value. The `Outputs` method in the block target file used to generate this code is

```
%function Outputs(block, system) Output
/* %<Type> Block: %<Name> */
  %%
  /* Multiply input by two */
  %assign rollVars = ["U", "Y"]
  %roll idx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
    %<LibBlockOutputSignal(0, "", lcv, idx)> = \
    %<LibBlockInputSignal(0, "", lcv, idx)> * 2.0;
  %endroll
%endfunction
```

Alter these temporary copies of the MATLAB file S-function and the TLC file to see how they interact. Start out by just changing the comments in the TLC file and see the changes appear in the generated code, then work up to algorithmic changes.

Inline Fortran (F-MEX) S-Functions

The capabilities of Fortran MEX S-functions can be fully inlined using a TLC block target file. This interface can be illustrated with a Fortran MEX S-function that implements the `timestwo` function. Here is the sample Fortran S-function code:

```
C
C     FTIMESTWO.FOR
C
C
C     A sample FORTRAN representation of a
C     timestwo S-function.
C     Copyright 1990-2000 The MathWorks, Inc.
C
C=====================================================
C     Function:  SIZES
C
C     Abstract:
C       Set the size vector.
C
C       SIZES returns a vector which determines model
C       characteristics.  This vector contains the
C       sizes of the state vector and other
C       parameters. More precisely,
C       SIZE(1)  number of continuous states
C       SIZE(2)  number of discrete states
C       SIZE(3)  number of outputs
C       SIZE(4)  number of inputs
C       SIZE(5)  number of discontinuous roots in
C                the system
C       SIZE(6)  set to 1 if the system has direct
C                feedthrough of its inputs,
C                otherwise 0
C
C=====================================================
      SUBROUTINE SIZES(SIZE)
C     .. Array arguments ..
      INTEGER*4        SIZE(*)
C     .. Parameters ..
      INTEGER*4       NSIZES
      PARAMETER       (NSIZES=6)

      SIZE(1) = 0
      SIZE(2) = 0
```

```
      SIZE(3) = 1
      SIZE(4) = 1
      SIZE(5) = 0
      SIZE(6) = 1

      RETURN
      END

C
C=======================================================
C     Function:  OUTPUT
C
C     Abstract:
C       Perform output calculations for continuous
C       signals.
C=======================================================
C     .. Parameters ..
      SUBROUTINE OUTPUT(T, X, U, Y)
      REAL*8           T
      REAL*8           X(*), U(*), Y(*)

      Y(1) = U(1) * 2.0

      RETURN
      END

C
C=======================================================
C     Stubs for unused functions.
C=======================================================

      SUBROUTINE INITCOND(XO)
      REAL*8           XO(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE DERIVS(T, X, U, DX)
      REAL*8           T, X(*), U(*), DX(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE DSTATES(T, X, U, XNEW)
```

```
      REAL*8           T, X(*), U(*), XNEW(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE DOUTPUT(T, X, U, Y)
      REAL*8           T, X(*), U(*), Y(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE TSAMPL(T, X, U, TS, OFFSET)
      REAL*8           T,TS,OFFSET,X(*),U(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE SINGUL(T, X, U, SING)
      REAL*8           T, X(*), U(*), SING(*)
C --- Nothing to do.
      RETURN
      END
```

Copy the preceding code into file `ftimestwo.for` in a convenient working folder.

Putting this into an S-function block in a simple model will illustrate the interface for inlining the S-function. Once your Fortran MEX environment is set up, prepare the code for use by compiling the S-function in a working folder along with the file `simulink.for` from *`matlabroot`*`/simulink/src/`.

This is done with the `mex` command at the MATLAB command line:

```
mex -fortran ftimestwo.for simulink.for
```

Now reference this block from a simple model set with a fixed-step solver and the `grt` target.

The TLC code for inlining this block is a modified form of `timestwo.tlc`. In your working folder, create a file named `ftimestwo.tlc` and put this code into it.

```
%implements "ftimestwo" "C"

%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %%
  /* Multiply input by two */
  %assign rollVars = ["U", "Y"]
  %roll idx = RollRegions, lcv = RollThreshold, block, ...
"Roller", rollVars
    %<LibBlockOutputSignal(0, "", lcv, idx)> = \
    %<LibBlockInputSignal(0, "", lcv, idx)> * 2.0;
  %endroll
%endfunction
```

Now you can generate code for the `ftimestwo` Fortran MEX S-function. The resulting code fragment specific to `ftimestwo` is

```
/* S-Function Block: <Root>/F-MEX S-Function */
  /* Multiply input by two */
  rtB.F_MEX_S_Function = rtB.Gain * 2.0;
```

TLC Coding Conventions

In this section...

Overview

These guidelines can help you apply programming style in each target file consistently.

Begin Identifiers with Uppercase Letters

Identifiers in the Simulink Coder file begin with an uppercase letter. For example,

```
NumModelInputs              1
NumModelOutputs             2
NumNonVirtBlocksInModel     42
DirectFeedthrough           yes
NumContStates               10
```

Because a `Name` identifier may be promoted into the parent scope, block records that contain a `Name` identifier should start the name with an uppercase letter. For example, a block might contain

```
Block {
   :
   :
RWork                  [4, 0]
   :
NumRWorkDefines        4
```

```
RWorkDefine {
    Name               "TimeStampA"
    Width              1
    StartIndex         0
  }
}
```

Because the `Name` identifier within the `RWorkDefine` record is promoted to `PrevT` in its parent scope, it must start with an uppercase letter. The promotion of the `Name` identifier into the parent block scope is currently done for the `Parameter`, `RWorkDefine`, `IWorkDefine`, and `PWorkDefine` block records.

The Target Language Compiler assignment directive (`%assign`) generates a warning if you assign a value to an "unqualified" Simulink Coder identifier. For example,

```
%assign TID = 1
```

produces an error because the `TID` identifier is not qualified by `Block`. However, a "qualified" assignment does not generate a warning. For example,

```
%assign Block.TID = 1
```

does not generate a warning because the assignment contains a qualifier. The Target Language Compiler therefore assumes that the programmer is intentionally modifying an identifier.

Begin Global Variable Assignments with Uppercase Letters

Global TLC variable assignments should start with uppercase letters. A global variable is a variable declared in a system target file (`grt.tlc`, `mdlwide.tlc`, `mdlhdr.tlc`, `mdlbody.tlc`, `mdlreg.tlc`, or `mdlparam.tlc`), or within a function that uses the operator. Global assignments have the same scope as Simulink Coder variables. An example of a global TLC variable defined in `mdlwide.tlc` is

```
%assign InlineParameters = 1
```

An example of a global reference in a function is

```
%function foo() void
  %assign ::GlobalIdx = ::GlobalIdx + 1
%endfunction
```

Begin Local Variable Assignments with Lowercase Letters

Local TLC variable assignments should start with lowercase letters. A local TLC variable is a variable assigned inside a function. For example,

```
%assign numBlockStates = ContStates[0]
```

Begin Functions Declared in block.tlc Files with Fcn

When you declare a function inside a `block.tlc` file, it should start with `Fcn`. For example,

```
%function FcnMyBlockFunc(...)
```

Note Functions declared inside a system file are global; functions declared inside a block file are local.

Do Not Hard-Code Variables Defined in commonsetup.tlc

Because the code generator tracks use of variables and generates code based on usage, you should use access routines instead of directly using a variable. For example, you should not use the following in your TLC file:

```
x = %<tInf>;
```

You should use

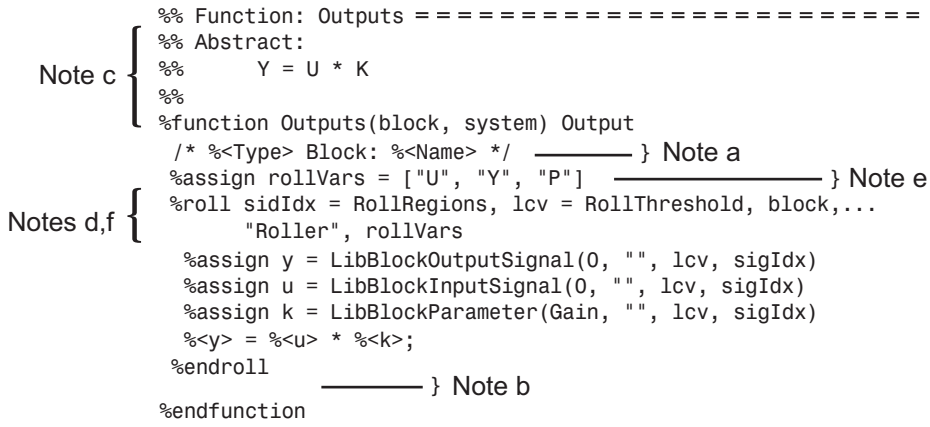
```
x = %<LibRealNonFinite(inf)>;
```

Similarly, instead of using `%<tTID>`, use `%<LibTID()>`. For a complete list of functions, see "TLC Function Library Reference".

Simulink CoderGlobal variables start with `rt` and Simulink Coder global functions start with `rt_`.

Avoid naming global variables in run-time interface modules that start with `rt` or `rt_` because they might conflict with Simulink Coder global variables and functions. These TLC variables are declared in `commonsetup.tlc`.

This convention creates consistent variables throughout the target files. For example, the Gain block contains the following `Outputs` function.



Notes about this TLC code

a. The code section for each block begins with a comment specifying the block type and name.

b. Include a blank line immediately after the end of the function to create consistent spacing between blocks in the output code.

c. Try to stay within 80 columns per line for the function banner. You might set up an 80 column comment line at the top of each function. As an example, see `constant.tlc`.

d. For consistency, use the variables `sysIdx` and `blkIdx` for system index and block index, respectively.

e. Use the variable `rollVars` when using the `%roll` construct.

f. When naming loop control variables, use `sigIdx` and `lcv` when looping over `RollRegions` and `xidx` and `xlcv` when looping over the states.

Example: `Output` function in `gain.tlc`

```
%roll sigIdx = RollRegions, lcv = RollThreshold, ...
       block, "Roller", rollVars
```

Example: `InitializeConditions` function in `linblock.tlc`

```
%roll xidx = [0:nStates-1], xlcv = RollThreshold,...
       block, "Roller", rollVars
```

Conditional Inclusion in Library Files

The Target Language Compiler function library files are conditionally included with guard code so you can reference them multiple times using `%include` without worrying if they have previously been included. Follow this practice for TLC library files that you create.

The convention is to use a variable with the same name as the base filename, uppercase and with underscores attached at both ends. So, a file named `customlib.tlc` should have the variable `_CUSTOMLIB_` guarding it.

As an example, the main Target Language Compiler function library, `funclib.tlc`, contains this TLC code to prevent multiple inclusion:

```
%if EXISTS("_FUNCLIB_") == 0
%assign _FUNCLIB_ = 1
  .
  .
  .
%endif  %% _FUNCLIB_
```

Code Defensively

As the code your TLC generates could be used in referenced models in unpredictable contexts, do not assume too much about name spaces. For example, when writing TLC code for a block and adding a `typedef`, guard it with `if/def`, as the following example illustrates:

```
%openfile tmpBuff
    #ifndef RESOLUTION_TYPEDEF

    typedef enum { LO_RES, HI_RES } Resolution;
    typedef struct { Resolution res; int8_T value; } Data;

    #define RESOLUTION_TYPEDEF
    #endif /* RESOLUTION_TYPEDEF */
  %closefile tmpBuff

  %<LibCacheTypedefs(tmpBuff)>;
```

Block Target File Methods

In this section...

Block Functions Overview

Each block has a target file that determines what code should be generated for the block. The code can vary depending on the exact parameters of the block or the types of connections to it (e.g., wide vs. scalar input).

Within each block target file, *block functions* specify the code to be output for the block in the model's or subsystem's `start` function, `output` function, `update` function, and so on.

The functions declared inside each of the block target files are called by the system target files. In these tables, `block` refers to a Simulink block name (e.g., `gain` for the Gain block) and `system` refers to the subsystem in which the block resides. The first table lists the two functions that are used for preprocessing and setup. Neither of these functions outputs generated code.

- "BlockInstanceSetup(block, system)" on page 8-29
- "BlockTypeSetup(block, system)" on page 8-30

The following functions generate executable code that the code generator places appropriately:

In object-oriented programming terms, these functions are polymorphic in nature, because each block target file contains the same functions. The Target Language Compiler dynamically determines at run-time which block function to execute depending on the block's type. That is, the system file only specifies that the `Outputs` function, for example, is to be executed. The particular `Outputs` function is determined by the Target Language Compiler depending on the block's type.

To write a block target file, use these polymorphic block functions combined with the Target Language Compiler library functions. For a complete list of the Target Language Compiler library functions, see "TLC Function Library Reference".

BlockInstanceSetup(block, system)

The `BlockInstanceSetup` function executes for the blocks that have this function defined in their target files in a model. For example, if a model includes 10 From Workspace blocks, then the `BlockInstanceSetup` function in `fromwks.tlc` executes 10 times, once for each From Workspace block instance. Use `BlockInstanceSetup` to generate code for each instance of a given block type.

See "TLC Function Library Reference" for available utility processing functions to call from inside this block function. See *`matlabroot`*`/rtw/c/tlc/blocks/lookup2d.tlc` for an example of the `BlockInstanceSetup` function.

Syntax

```
BlockInstanceSetup(block, system) void
   block = Reference to a Simulink block
   system = Reference to a nonvirtual Simulink subsystem
```

This example uses `BlockInstanceSetup`:

```
%function BlockInstanceSetup(block, system) void
%if (block.InMask == "yes")
    %assign blockName = LibParentMaskBlockName(block)
  %else
    %assign blockName = LibGetFormattedBlockPath(block)
  %endif
  %if (CodeFormat == "Embedded-C")
    %if !(ParamSettings.ColZeroTechnique == "NormalInterp" && ...
          ParamSettings.RowZeroTechnique == "NormalInterp")
      %selectfile STDOUT

Note: Removing repeated zero values from the X and Y axes will
produce more efficient code for block: %<blockName>.  To locate
this block, type

open_system('%<blockName>')

at the MATLAB command prompt.

      %selectfile NULL_FILE
    %endif
  %endif
%endfunction
```

BlockTypeSetup(block, system)

`BlockTypeSetup` executes once per block type before code generation begins. That is, if 10 Lookup Table blocks exist in the model, the `BlockTypeSetup` function in `look_up.tlc` is called only one time. Use this function to perform general work for multiple blocks of a given type.

See "TLC Function Library Reference" for a list of relevant functions to call from inside this block function. See `look_up.tlc` for an example of the `BlockTypeSetup` function.

Syntax

```
BlockTypeSetup(block, system) void
   block = Reference to a Simulink block
   system = Reference to a nonvirtual Simulink subsystem
```

As an example, given the S-function `foo`, which requires a `#define` and two function declarations in the header file, you could define:

```
%function BlockTypeSetup(block, system) void

  %% Place a #define in the model's header file

  %openfile buffer
    #define A2D_CHANNEL 0
  %closefile buffer

  %<LibCacheDefine(buffer)>

  %% Place function prototypes in the model's header file

  %openfile buffer
    void start_a2d(void);
    void reset_a2d(void);
  %closefile buffer

  %<LibCacheFunctionPrototype(buffer)>
%endfunction
```

The remaining block functions execute once for each block in the model.

Enable(block, system)

The code generator creates `Enable` functions for nonvirtual subsystem whenever a Simulink subsystem contains a block with an `Enable` function. Including the `Enable` function in a block's target file places the block's specific enable code in this subsystem `Enable` function. For example:

```
%% Function: Enable ===========================================
%% Abstract:
%% Subsystem Enable code is required only for the discrete form
%% of the Sine Block. Setting the Boolean to TRUE causes the
%% Output function to resync its last values of cos(wt) and
%% sin(wt).
%%
%function Enable(block, system) Output
  %if LibIsDiscrete(TID)
    /* %<Type> Block: %<Name> */
    %<LibBlockIWork(SystemEnable, "", "", 0)> = (int_T) TRUE;

  %endif
%endfunction
```

Disable(block, system)

Nonvirtual subsystem `Disable` functions are created whenever a Simulink subsystem contains a block with a `Disable` function. Including the `Disable` function in a block's target file places the block's specific disable code into this subsystem `Disable` function. See `outport.tlc` in *`matlabroot`*`/rtw/c/tlc/blocks` for an example of the `Disable` function.

Start(block, system)

Include a `Start` function to place code in the `Start` function. The code inside the `Start` function executes once and only once. Typically, you include a `Start` function to execute code once at the beginning of the simulation (e.g., initialize values in the work vectors) or code that does not need to be re-executed when the subsystem in which it resides is enabled. See `constant.tlc` for an example of the `Start` function.

```
%% Function: Start ===========================================
%% Abstract:
%% Set the output to the constant parameter value if the block
%% output is visible in the model's start function scope, i.e.,
%% it is in the global rtB structure.
%%
%function Start(block, system) Output
  %if  LibBlockOutputSignalIsInBlockIO(0)
    /* %<Type> Block: %<Name> */
    %assign rollVars = ["Y", "P"]
    %roll idx = RollRegions, lcv = RollThreshold, block, ...
      "Roller", rollVars
      %assign yr = LibBlockOutputSignal(0,"", lcv, ...
        "%<tRealPart>%<idx>")
      %assign pr = LibBlockParameter(Value, "", lcv, ...
        "%<tRealPart>%<idx>")
      %<yr> = %<pr>;
      %if LibBlockOutputSignalIsComplex(0)
        %assign yi = LibBlockOutputSignal(0, "", lcv, ...
          "%<tImagPart>%<idx>")
        %assign pi = LibBlockParameter(Value, "", lcv, ...
          "%<tImagPart>%<idx>")
        %<yi> = %<pi>;
      %endif
    %endroll
  %endif
```

```
%endfunction %% Start
```

InitializeConditions(block, system)

TLC code that is generated from the block's `InitializeConditions` function appears in one of two places. A nonvirtual subsystem contains an `Initialize` function when it is configured to reset states on enable. In this case, the TLC code generated by this block function is placed in the subsystem `Initialize` function, and the `start` function calls this subsystem `Initialize` function. If, however, the Simulink block resides in the root system or in a nonvirtual subsystem that does not require an `Initialize` function, the code generated from this block function is placed directly (inlined) into the `start` function.

There is a subtle difference between the block functions `Start` and `InitializeConditions`. Typically, you include a `Start` function to execute code that does not need to re-execute when the subsystem in which it resides is enabled. You include an `InitializeConditions` function to execute code that must re-execute when the subsystem in which it resides is enabled. For example:

```
%% Function: InitializeConditions ============================
%%
%% Abstract: Invalidate the stored output and input in
%% rwork[1 2*blockWidth] by setting the time stamp stored
%% in rwork[0]) to rtInf.
%%
%function InitializeConditions(block, system) Output
  /* %<Type> Block: %<Name> */
  %<LibBlockRWork(PrevT, "", "", 0)> = %<LibRealNonFinite(inf)>;
%endfunction
```

Outputs(block, system)

A block should generally include an `Outputs` function. The TLC code generated by a block's `Outputs` function is placed in one of two places. The code is placed directly in the model's `Outputs` function if the block does not reside in a nonvirtual subsystem, and in a subsystem's `Outputs` function if the block resides in a nonvirtual subsystem. For example:

```
%% Function: Outputs =========================================
%% Abstract:
%%      Y[i] = fabs(U[i]) if U[i] is real or
```

```
%%      Y[i] = sqrt(U[i].re^2 + U[i].im^2) if U[i] is complex.
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %%
  %assign inputIsComplex = LibBlockInputSignalIsComplex(0)
  %assign RT_SQUARE = "RT_SQUARE"
  %%
  %assign rollVars = ["U", "Y"]
  %if inputIsComplex
    %roll sigIdx = RollRegions, lcv = RollThreshold, ...
      block, "Roller", rollVars
      %%
      %assign ur = LibBlockInputSignal( 0, "", lcv, ...
        "%<tRealPart>%<sigIdx>")
      %assign ui = LibBlockInputSignal( 0, "", lcv, ...
        "%<tImagPart>%<sigIdx>")
      %%
      %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
      %<y> = sqrt( %<RT_SQUARE>( %<ur> ) + %<RT_SQUARE>( %<ui> ) );
    %endroll
  %else
    %roll sigIdx = RollRegions, lcv = RollThreshold, ...
      block, "Roller", rollVars
      %assign u = LibBlockInputSignal (0, "", lcv, sigIdx)
      %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
      %<y> = fabs(%<u>);
    %endroll
  %endif
  %endfunction
```

Note Zero-crossing reset code is placed in the `Outputs` function.

Update(block, system)

Include an `Update` function if the block has code that needs to be updated at each major time step. Code generated from this function is placed in either the model's or the subsystem's `Update` function, depending on whether or not the block resides in a nonvirtual subsystem. For example:

```
%% Function: Update ============================================
%% Abstract:
```

```
%%      X[i] = U[i]
%%
%function Update(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign rollVars = ["U", "Xd"]
  %roll idx = RollRegions, lcv = RollThreshold, block, ...
    "Roller", rollVars
    %assign u = LibBlockInputSignal(0, "", lcv, idx)
    %assign x = LibBlockDiscreteState("", lcv, idx)
    %<x> = %<u>;
  %endroll
%endfunction %% Update
```

Derivatives(block, system)

Include a `Derivatives` function when generating code to compute the block's continuous states. Code generated from this function is placed in either the model's or the subsystem's `Derivatives` function, depending on whether or not the block resides in a nonvirtual subsystem. See `integrat.tlc` for an example of the `Derivatives` function.

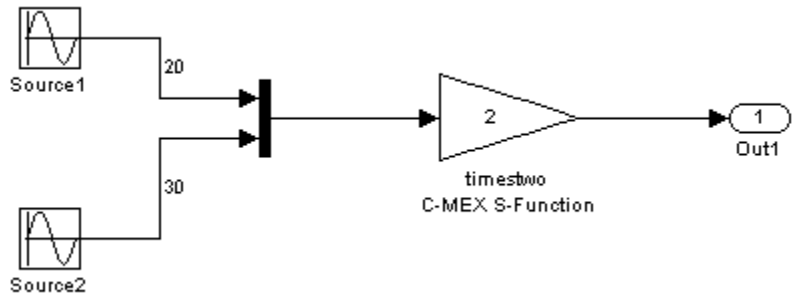
Terminate(block, system)

Include a `Terminate` function to place code in `MdlTerminate`. User-defined S-function target files can use this function to save data, free memory, reset hardware on the target, and so on. See `tofile.tlc` for an example of the `Terminate` function.

Loop Rolling

One of the optimization features of the Target Language Compiler is the intrinsic support for loop rolling. Based on a specified threshold, code generation for looping operations can be unrolled or left as a loop (rolled).

Coupled with loop rolling is the concept of noncontiguous signals. Consider the following model:



The input to the `timestwo` S-function comes from two arrays located at two different memory locations, one for the output of `source1` and one for the output of block `source2`. This is because of an optimization that makes the Mux block *virtual*, meaning that code is not explicitly generated for the Mux block and thus processor cycles are not spent evaluating it (i.e., it becomes a pure graphical convenience for the block diagram). So this is represented in the *`model`*`.rtw` file in this case as

```
Block {
      Type         "S-Function"
      MaskType         "S-function: timestwo"
      BlockIdx         [0, 0, 2]
      SL_BlockIdx        2
      GrSrc         [0, 1]
      ExprCommentInfo {
 SysIdxList  []
 BlkIdxList  []
 PortIdxList  []
      }
      ExprCommentSrcIdx {
 SysIdx   -1
 BlkIdx   -1
```

```
PortIdx   -1
     }
     Name         "<Root>/timestwo  C-MEX S-Function"
     SLName         "<Root>/timestwo \nC-MEX S-Function"
     Identifier        timestwoCMEXSFunction
     TID        0
     RollRegions       [0:19, 20:49]
     NumDataInputPorts       1
     DataInputPort {
SignalSrc  [b0@20, b1@30]
SignalOffset  [0:19, 0:29]
Width   50
RollRegions  [0:19, 20:49]
     }
     NumDataOutputPorts      1
     DataOutputPort {
SignalSrc  [b2@50]
SignalOffset  [0:49]
Width   50
     }
     Connections {
InputPortContiguous [no]
InputPortConnected [yes]
OutputPortConnected [yes]
OutputPortBeingMerged [no]
DirectSrcConn  [no]
DirectDstConn  [yes]
DataOutputPort {
  NumConnPoints    1
  ConnPoint {
    SrcSignal      [0, 50]
    DstBlockAndPortEl     [0, 4, 0, 0]
  }
}
 }
.
.
.
```

From this fragment of the *model*`.rtw` file you can see that the block and input port `RollRegion` entries are not just one number, but two groups of numbers. This denotes two groupings in memory for the input signal. The generated code looks like this:

```
/* S-Function Block: <Root>/timestwo  C-MEX S-Function */
```

```
  /* Multiply input by two */
  {
    int_T i1;

    const real_T *u0 = &contig_sample_B.u[0];
    real_T *y0 = contig_sample_B.timestwoCMEXSFunction_m;

    for (i1=0; i1 < 20; i1++) {
      y0[i1] = u0[i1] * 2.0;
    }

    u0 = &contig_sample_B.u_o[0];
    y0 = &contig_sample_B.timestwoCMEXSFunction_m[20];

    for (i1=0; i1 < 30; i1++) {
      y0[i1] = u0[i1] * 2.0;
    }
}
```

Notice that two loops are generated and between them the input signal is redirected from the first base address, `&contig_sample_B.u[0]`, to the second base address of the signals, `&contig_sample_B.u_o[0]`. If you do not want to support this in your S-function or your generated code, you can use

```
ssSetInputPortRequiredContiguous(S, 1);
```

in the `mdlInitializeSizes` function to cause Simulink to implicitly generate code that performs a buffering operation. This option uses both extra memory and CPU cycles at run-time, but might be worth it if your algorithm performance increases enough to offset the overhead of the buffering.

Use the `%roll` directive to generate loops. See also "%roll" on page 6-29 for the reference entry for `%roll`, and "Input Signal Functions" on page 9-8 for a discussion on the behavior of `%roll`.

Error Reporting

You might need to detect and report error conditions in your TLC code. Error detection and reporting are used most often in library functions. While rare, it is also possible to encounter error conditions in block target file code if the S-function `mdlCheckParameters` function does not detect an unforeseen condition.

To report an error condition detected in your TLC code, use the `LibBlockReportError` or `LibBlockReportFatalError` utility functions. Here is an example of using `LibBlockReportError` in the `paramlib.tlc` function `LibBlockParameter` to report the condition of an improper use of that function:

```
%if TYPE(param.Value) == "Matrix"
    %% exit if the parameter is a true matrix,
    %% i.e., has more than one row or columns.
    %if nRows > 1
      %assign errTxt = "Must access parameter %<param.Name> using "...
        "LibBlockMatrixParameter."
      %<LibBlockReportError([], errTxt)>
    %endif
  %endif
```

Browse through *`matlabroot`*`/rtw/c/tlc` for more examples of the use of `LibBlockReportError`. Also, read further details in “Error Handling”, which describes types of TLC errors and their interpretations.

TLC Function Library Reference

This chapter provides a set of Target Language Compiler functions that are useful for inlining S-functions. The TLC files contain many other library functions, but you should use only the functions that are documented in these reference pages for development. Undocumented functions can change significantly from release to release. "Obsolete Functions" on page 9-2 includes a table of obsolete functions and their replacements.

You can find examples using these functions in `matlabroot/toolbox/simulink/blocks/tlc_c` and `matlabroot/toolbox/simulink/simdemos/simfeatures/tlc_c`. The corresponding MEX S-function source code is located in `matlabroot/simulink/src` or `matlabroot/toolbox/simulink/simdemos/simfeatures/src`. MATLAB file S-functions and the MEX-file executables (for example, *`sfunction`*`.mex*`) are located in `matlabroot/toolbox/simulink/blocks` or `matlabroot/toolbox/simulink/simdemos/simfeatures`.

Obsolete Functions

The following table shows obsolete functions and the functions that have replaced them.

Obsolete Function	Equivalent Replacement Function
`LibBlockOutportLocation`	`LibBlockDstSignalLocation`
`LibCacheGlobalPrmData`	Use the block function `Start`
`LibCacheIncludes`	`LibAddToCommonIncludes`
`LibContinuousState`	`LibBlockContinuousState`
`LibControlPortInputSignal`	`LibBlockSrcSignalLocation`
`LibConvertZCDirection`	Function is not used in Simulink Coder code generation.
`LibDataInputPortWidth`	`LibBlockInputSignalWidth`
`LibDataOutputPortWidth`	`LibBlockOutputSignalWidth`

`LibDefineIWork`
`LibDefinePWork`
`LibDefineRWork`
	`IWork` , `PWork`, and `RWork` names are now specified via the `mdlRTW` function in your C MEX S-function.

`LibDiscreteState`	`LibBlockDiscreteState`
`LibExportFileCustomCode`	`LibSetSourceFileSection`
`LibExternalResetSignal`	`LibBlockInputSignal`
`LibHeaderFileCustomCode`	`LibSetSourceFileSection`
`LibIsEqual`	Use built-in function `ISEQUAL`
`LibMapSignalSource`	`FcnMapDataTypedSignalSource`
`LibMaxBlockIOWidth`	Function is not used in Simulink Coder code generation.
`LibMaxDataInputPortWidth`	Function is not used in Simulink Coder code generation.
`LibMaxDataOutputPortWidth`	Function is not used in Simulink Coder code generation.
`LibPathName`	`LibGetBlockPath`, `LibGetFormattedBlockPath`
`LibPrevZCState`	`LibBlockPrevZCState`

Obsolete Function	Equivalent Replacement Function
`LibPrmFileCustomCode`	`LibSetSourceFileSection`
`LibRegFileCustomCode`	`LibSetSourceFileSection`
`LibRegisterGNUMathFcnPrototypes`	Function is not used in Simulink Coder code generation.
`LibRegisterISOCMathFcnPrototypes`	Function is not used in Simulink Coder code generation.
`LibRegisterMathFcnPrototype`	Function is not used in Simulink Coder code generation.
`LibRenameParameter`	Specifying parameter names is now supported via the `mdlRTW` function in your C MEX S-function.

Target Language Compiler Function Conventions

In this section...

Common Function Arguments

Several functions take similar or identical arguments. To simplify the reference pages, some of these arguments are documented in detail here instead of in the reference pages.

Argument	Description
`portIdx`	Refers to an input or output port index, starting at 0. For example, the first input port of an S-function is 0.
`ucv`	User control variable. This is an advanced feature that overrides the `lcv` and `sigIdx` parameters. When used within an inlined S-function, it should generally be specified as `""`.
`lcv`	Loop control variable. This is generally generated by the `%roll` directive via the second `%roll` argument (e.g., `lcv=RollThreshold`) and should be passed directly to the library function. It contains either `""`, indicating that the current pass through the `%roll` is being inlined, or it is the name of a loop control variable such as `"i"`, indicating that the current pass through the `%roll` is being placed in a loop. Outside the `%roll` directive, this is usually specified as `""`.
`sigIdx`*or*`idx`	Signal index. Sometimes referred to as the signal element index. When accessing specific elements of an input or output signal directly, the call to the various library routines should have `ucv=""`, `lcv=""`, and `sigIdx` equal to the desired integer signal index starting at 0. For complex signals, `sigIdx` can be an overloaded integer index specifying both whether the real or imaginary part is being accessed and which element. When you access these items inside a `%roll`, use the `sigIdx` generated by the `%roll` directive. Most functions that take a `sigIdx` argument accept it in an overloaded form, where `sigIdx` can be • An integer, e.g., `3`. If the referenced signal is complex, then this refers to the identifier for the complex container. If the referenced signal is not complex, then this refers to the identifier.

Argument	Description
	• An `id-num`, usually of the form (see "Overloading sigIdx" on page 9-5) **a** `"%<tRealPart>%<idx>"` (e.g., `"re3"`). The real part of the signal element. Usually `"%<tRealPart>%<sigIdx>"` when `sigIdx` is generated by the `%roll` directive. **b** `"%<tImagPart>%<idx>"` (e.g., `"im3"`). The imaginary part of the signal element or `""` if the signal is not complex. Usually `"%<tImagPart>%<sigIdx>"` when `sigIdx` is generated by the `%roll` directive. Use the `idx` name when referring to a state or work vector. Functions that accept the three arguments `ucv`, `lcv`, `sigIdx` (or `idx`) are called differently depending upon whether or not they are used within a `%roll` directive. If they are used within a `%roll` directive, `ucv` is generally specified as `""` and `lcv` and `sigIdx` are the same as those specified in the `%roll` directive. If they are not used within a `%roll` directive, `ucv` and `lcv` are generally specified as `""`, and `sigIdx` specifies the index to access.
`paramIdx`	Parameter index. Sometimes referred to as the parameter element index. The handling of this parameter is very similar to `sigIdx` above: it can be `#`, `re#`, or `im#`.
`stateIdx`	State index. Sometimes referred to as the state vector element index. It must evaluate to an integer where the first element starts at `0`.

Overloading sigIdx

The signal index (`sigIdx` sometimes written as `idx`) can be overloaded when passed to most library functions. Suppose you are interested in element 3 of a signal, and `ucv=""`, `lcv=""`. The following table shows

- Values of `sigIdx`
- Whether the signal being referenced is complex
- What the function that uses `sigIdx` returns
- An example of a returned variable
- Data type of the returned variable

Note that "container" in the following table refers to the object that encapsulates both the real and imaginary parts of the number, e.g., `creal_T`, defined in *`matlabroot`*`/extern/include/tmwtypes.h`.

sigIdx	Complex	Function Returns	Example	Data Type
`"re3"`	Yes	Real part of element 3	`u0[2].re`	`real_T`
`"im3"`	Yes	Imaginary part of element 3	`u0[2].im`	`real_T`
`"3"`	Yes	Complex container of element 3	`u0[2]`	`creal_T`
`3`	Yes	Complex container of element 3	`u0[2]`	`creal_T`
`"re3"`	No	Element 3	`u0[2]`	`real_T`
`"im3"`	No	`""`	N/A	N/A
`"3"`	No	Element 3	`u0[2]`	`real_T`
`3`	No	Element 3	`u0[2]`	`real_T`

Now suppose the following:

1. You are interested in element 3 of a signal.
2. (`ucv = "i"` AND `lcv == ""`) OR (`ucv = ""` AND `lcv = "i"`).

The following table shows values of `idx`, whether the signal is complex, and what the function that uses `idx` returns.

sigIdx	Complex	Function Returns
`"re3"`	Yes	Real part of element `i`
`"im3"`	Yes	Imaginary part of element `ii`
`"3"`	Yes	Complex container of element `i`
`3`	Yes	Complex container of element `i`
`"re3"`	No	Element `i`
`"im3"`	No	`""`
`"3"`	No	Element `i`

sigIdx	Complex	Function Returns
3	No	Element `i`

Notes

- The vector index is added only for wide signals.
- If `ucv` is not an empty string (`""`), then `ucv` is used instead of `sigIdx` in the above examples and both `lcv` and `sigIdx` are ignored.
- If `ucv` is empty but `lcv` is not empty, then the function returns `"&y%<portIdx>[%<lcv>]"` and `sigIdx` is ignored.
- It is assumed that the roller has declared and initialized the variables accessed inside the roller. The variables accessed inside the roller should be specified using `rollVars` as the argument to the `%roll` directive.

Input Signal Functions

In this section...

LibBlockInputPortIndexMode(block, pidx)

Purpose

Determines the index mode of a block's input port.

Arguments

`block` — Block record

`pidx` — Port index

Returns

`""` for a nonindex port, and `"Zero-based"` or `"One-based"` otherwise.

Description

If an input port of a `block` is set as an index port and its indexing base is marked as zero-based or one-based, this information is written into the *`model`*`.rtw` file. `LibBlockInputPortIndexMode` queries the indexing base to branch to different code according to what the input port indexing base is.

Example

```
%if LibBlockInputPortIndexMode(block, pidx) == "Zero-based"
  ...
%elseif LibBlockInputPortIndexMode(block, pidx) == "One-based"
  ...
%else
  ...
%endif
```

See `LibBlockInputPortIndexMode` in `matlabroot/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignal(portIdx, ucv, lcv, sigIdx)

Based on the input port number (`portIdx`), the user control variable (`ucv`), the loop control variable (`lcv`), the signal index (`sigIdx`), and where this input signal is coming from, `LibBlockInputSignal` returns the reference to a block input signal.

The returned string value is a valid `rvalue` (right-side value) for an expression. The block input signal can come from another block, a state vector, or an external input, or it can be a literal constant (e.g., 5.0).

Note Do not use `LibBlockInputSignal` to access the address of an input signal.

Because the returned value can be a literal constant, you should not use `LibBlockInputSignal` to access the address of an input signal. To access the address of an input signal, use `LibBlockInputSignalAddr`. Accessing the address of the signal via `LibBlockInputSignal` can result in a reference to a literal constant (e.g., 5.0).

For example, the following would *not* work.

```
%assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
x = &%<u>;
```

If %<u> refers to an invariant signal with a value of 4.95, the statement (after being processed by the preprocessor) would be generated as

```
x = &4.95;
```

or, if the input signal sources to ground, the statement could come out as

```
x = &0.0;
```

Neither of these would compile.

Avoid such situations by using LibBlockInputSignalAddr.

```
%assign uAddr = LibBlockInputSignalAddr(0, "", lcv, sigIdx)
x = %<uAddr>;
```

The Simulink Coder code generator tracks signals and parameters accessed by their addresses and declares them in addressable memory.

Input Arguments

The following table summarizes the input arguments to LibBlockInputSignal.

LibBlockInputSignal Arguments

Argument	Description
portIdx	Integer specifying the input port index (zero-based). **Note:** For certain built-in blocks, portIdx can be a string identifying the port (such as "enable" or "trigger").
ucv	User control variable. Must be a string, either an indexing expression or "".
lcv	Loop control variable. Must be a string, either an indexing expression or "".
sigIdx	Either an integer literal or a string of the form %<tRealPart>Integer %<tImagPart>Integer For example, the following signifies the real part of the signal and the imaginary part of the signal starting at 5: "%<tRealPart>5" "%<tImagPart>5"

General Usage

Uses of `LibBlockInputSignal` fall into the categories described below.

Direct indexing

If `ucv == ""` and `lcv == ""`, `LibBlockInputSignal` returns an indexing expression for the element specified by `sigIdx`.

Loop rolling/unrolling

In this case, `lcv` and `sigIdx` are generated by the `%roll` directive, and `ucv` must be `""`. A nonempty value for `lcv` is allowed only when generated by the `%roll` directive and when using the Roller TLC file (or a user supplied Roller TLC file that conforms to the same variable/signal offset handling). In addition, calls to `LibBlockInputSignal` with `lcv` should occur only when `"U"` or a specific input port (e.g., `"u0"`) is passed to the `%roll` directive via the roll variables argument.

The following example shows a single input/single output port S-function.

```
%assign rollVars  = ["U", "Y", "P"]
%roll sigIdx=RollRegions, lcv=RollThreshold, block, ...
    "Roller", rollVars
  %assign u = LibBlockInputSignal( 0, "", lcv, sigIdx)
  %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
  %assign p = LibBlockParameter(   0, "", lcv, sigIdx)
  %<y> = %<p> * %<u>;
%endroll
```

With the `%roll` directive, `sigIdx` is the starting index of the current roll region and `lcv` is `""` or an indexing variable. The following are examples of valid values:

```
LibBlockInputSignal(0, "", lcv, sigIdx)    rtB.blockname[0]

LibBlockInputSignal(0, "", lcv, sigIdx)    u[i]
```

In the first example, `LibBlockInputSignal` returns `rtB.blockname[2]` when the input port is connected to the output of another block, and

- The loop control variable (`lcv`) generated by the `%roll` directive is empty, indicating that the current roll region is below the roll threshold, and `sigIdx` is `0`.
- The width of the input port is `1`, indicating that this port is being scalar expanded.

  If `sigIdx` is nonzero, then `rtB.blockname[sigIdx]` is returned. For example, if `sigIdx` is `3`, then `rtB.blockname[3]` is returned.

In the second example, `LibBlockInputSignal` returns `u[i]` when the current roll region is above the roll threshold and the input port width is nonscalar (wide). In this case, the Roller TLC file sets up a local variable, `u`, to point to the input signal, and the code in the current `%roll` directive is placed within a `for` loop.

For another example, consider a block with multiple input ports where each port has a width greater than or equal to 1 and at least one port has width equal to 1. The following code sets the output signal to the sum of the squares of the input signals.

```
%assign y = LibBlockOutputSignal(0, "", "", 0)
%<y> = 0;

%assign rollVars = ["U"]
%foreach port = block.NumDataInputPorts - 1
  %roll sigIdx=RollRegions, lcv = RollThreshold, block, ...
    "Roller", rollVars
  %assign u = LibBlockInputSignal(port, "", lcv, sigIdx)
  %<y> += %<u> * %<u>;
  %endroll
%endforeach
```

Because the first parameter of `LibBlockInputSignal` is `0` indexed, you must index the `foreach` loop to start from `0` and end at `NumDataInputPorts-1`.

User Control Variable (ucv) Handling

This is an advanced mode and generally not required by S-function authors.

If `ucv != ""`, `LibBlockInputSignal` returns an `rvalue` for the input signal using the user control variable indexing expression. The control variable indexing expression has the following form:

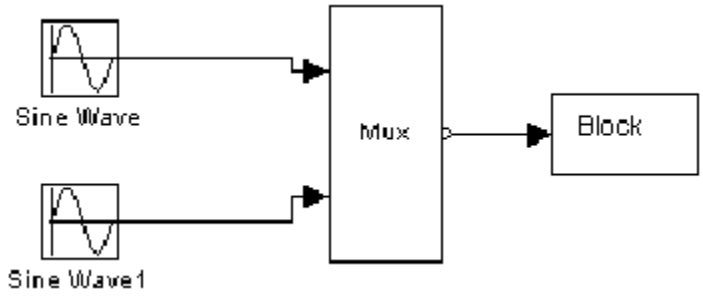
```
rvalue_id[%<ucv>]%<optional_real_or_imag_part>
```

To obtain `rvalue_id`, look at the integer part of `sigIdx`. You must specify `sigIdx` because the input to this block can be discontinuous, meaning that the input can come from several different memory areas (signal sources) and `sigIdx` is used to identify the area of interest for the `ucv`. You can also use `sigIdx` to determine whether the real or imaginary part of a signal is to be accessed.

You can obtain `optional_real_or_imag_part` from the string part of `sigIdx` (i.e., `"re"`, or `"im"`, or `""`).

Note that the value for `lcv` is ignored and `sigIdx` must point to the same element in the input signal to which the `ucv` initially points.

The handling of `ucv` with `LibBlockInputSignal` requires care. Consider a discontinuous input signal feeding an input port as in the following block diagram:



To use `ucv` in a robust manner, you must use the `%roll` directive with a roll threshold of `1` and a Roller TLC file that does not have loop header/trailer setup for this input signal. In addition, you need to use `ROLL_ITERATIONS` to determine the width of the current roll region, as in the following TLC code:

```
{
int i;

%assign rollVars  = [""]
%assign threshold = 1
  %roll sigIdx=RollRegions, lcv=threshold, block, ...
    "FlatRoller", rollVars
  %assign u = LibBlockInputSignal( 0, "i", "", sigIdx)
  %assign y = LibBlockOutputSignal(0, "i+%<sigIdx>", "", sigIdx)
  %assign p = LibBlockParameter( 0, "i+%<sigIdx>", "", sigIdx)
  for (i = 0; i < %<ROLL_ITERATIONS()>; i++) {
    %<y> = %<p> * %<u>;
  }
%endroll
}
```

Note that the `FlatRoller` does not have loop header/trailer setup (`rollVars` is ignored). Its purpose is to walk the `RollRegions` of the block. Alternatively, you can force a contiguous input signal to your block by specifying

```
ssSetInputPortRequiredContiguous(S, port, TRUE)
```

in your S-function.

In this case, the TLC code simplifies to

```
{
%assign u = LibBlockInputSignal( 0, "i", "", 0)
%assign y = LibBlockOutputSignal(0, "i", "", 0)
%assign p = LibBlockParameter(   0, "i", "", 0)

for (i = 0; i < %<LibBlockInputSignalWidth(0)>; i++) {
  %<y> = %<p> * %<u>;
  }
}
```

If you create your own roller and the indexing does not conform to the way the Roller TLC file provided by MathWorks operates, then must to use `ucv` instead of `lcv`.

Handling Input Arguments: ucv, lcv, and sigIdx

Consider the following cases:

Function (Case 1, 2, 3,4)	Example Return Value
`LibBlockInputSignal(0, "i", "", sigIdx)`	`rtB.blockname[i]`
`LibBlockInputSignal(0, "i", "", sigIdx)`	`rtU.signame[i]`
`LibBlockInputSignal(0, "", lcv, sigIdx)`	`u0[i1]`
`LibBlockInputSignal(0, "", lcv, sigIdx)`	`rtB.blockname[0]`

The value returned depends on what the input signal is connected to in the block diagram and how the function is invoked (e.g., in a `%roll` or directly). In the above example,

- Cases 1 and 2 occur when an explicit call is made with the `ucv` set to `"i"`.

  Case 1 occurs when `sigIdx` points to the block I/O vector, i.e., the first element that `"i"` starts with. For example, if you initialize `"i"` to be starting at offset `5`, then you should specify `sigIdx == 5`.

  Case 2 occurs when `sigIdx` points to the external input vector, i.e., the first element that `"i"` starts with. For example, if you initialize `"i"` to start at offset `20`, then you should specify `sigIdx == 20`.

- Cases 3 and 4 receive the same arguments, `lcv` and `sigIdx`; however, they produce different return values.

  Case 3 occurs when `LibBlockInputSignal` is called within a `%roll` directive and the current roll region is being rolled (`lcv != ""`).

  Case 4 occurs when `LibBlockInputSignal` is called within a `%roll` directive and the current roll region is not being rolled (`lcv == ""`).

When called within a`%roll` directive, `LibBlockInputSignal` looks at `ucv`, `lcv`, and `sigIdx`, the current roll region, and the current roll threshold to determine the return value. The variable `ucv` has highest precedence, `lcv` has the next highest precedence, and `sigIdx` has the lowest precedence. That is, if `ucv` is specified, it is used (thus, when called in a `%roll` directive it is usually `""`). If `ucv` is not specified and `lcv` and `sigIdx` are specified, the returned value depends on whether or not the current roll region is being placed in a `for` loop or being expanded. If the roll region is being placed in a loop, then `lcv` is used; otherwise, `sigIdx` is used.

A direct call to `LibBlockInputSignal` (inside or outside a `%roll` directive) uses `sigIdx` when `ucv` and `lcv` are specified as `""`.

For an example of `LibBlockInputSignal`, see `matlabroot/toolbox/simulink/simdemos/simfeatures/tlc_c/sfun_multiport.tlc`.

See also matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalAddr(portIdx, ucv, lcv, sigIdx)

Returns a string that provides the memory address of the specified block input port signal.

When you need an input signal address, you must use `LibBlockInputSignalAddr` instead of appending an "`&`" to the string returned by `LibBlockInputSignal`. For example, `LibBlockInputSignal` can return a literal constant, such as `5` (i.e., an invariant input signal). The Simulink Coder code generator tracks when `LibBlockInputSignalAddr` is called on an invariant signal and declares the signal as `const` data (which is addressable), instead of being placed as a literal constant in the generated code (which is not addressable).

Note that the last input argument, `sigIdx`, is not overloaded, which it is in `LibBlockInputSignal`. Hence, if the input signal is complex, the address of the complex container is returned.

Example

To get the address of a wide input signal and pass it to a user function for processing, you could use

```
%assign uAddr = LibBlockInputSignalAddr(0, "", "", 0)
%assign y = LibBlockOutputSignal(0, "", "", 0)
%<y> = myfcn(%<uAddr>);
```

See `LibBlockInputSignalAddr` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalAliasedThruDataTypeName (portIdx, reim)

Returns the name of the aliased thru data type (e.g., `int_T, ... creal_T`) corresponding to the specified block input port. Specify the `reim` argument as `""` (empty) if you want the complete signal type name.

For example, if `reim == ""` and the first output port is real and complex, the data type name placed in `dtname` is `creal_T`.

```
%assign dtname = LibBlockInputSignalDataTypeName(0,"")
```

Specify `reim` as `tRealPart` if you want the raw element type name. For example, if `reim == tRealPart` and the first output port is real and complex, the data type name returned is `real_T`.

```
%assign dtname = LibBlockOutputSignalDataTypeName(0,tRealPart)
```

See `LibBlockInputSignalAliasedThruDataTypeName` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalConnected(portIdx)

Returns 1 if the specified input port is connected to a block other than the Ground block and 0 otherwise.

See `LibBlockInputSignalConnected` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalDataTypeId(portIdx)

Returns the numeric identifier (`id`) corresponding to the data type of the specified block input port.

If the input port signal is complex, `LibBlockInputSignalDataTypeId` returns the data type of the real part (or the imaginary part) of the signal.

See `LibBlockInputSignalDataTypeId` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalDataTypeName(portIdx, reim)

Returns the name of the data type (e.g., `int_T`, ... `creal_T`) corresponding to the specified block input port.

Specify the `reim` argument as `""` if you want the complete signal type name. For example, if `reim==""` and the first output port is real and complex, the data type name placed in `dtname` is `creal_T`.

```
%assign dtname = LibBlockInputSignalDataTypeName(0,"")
```

Specify the `reim` argument as `tRealPart` if you want the raw element type name. For example, if `reim==tRealPart` and the first output port is real and complex, the data type name returned is `real_T`.

```
%assign dtname = LibBlockInputSignalDataTypeName(0,tRealPart)
```

See `LibBlockInputSignalDataTypeName` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalDimensions(portIdx)

Returns the dimensions vector of the specified block input port, e.g., `[2,3]`.

See `LibBlockInputSignalDimensions` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalIsComplex(portIdx)

Returns 1 if the specified block input port is complex, 0 otherwise.

See `LibBlockInputSignalIsComplex` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalIsFrameData(portIdx)

Returns 1 if the specified block input port is frame based, 0 otherwise.

See `LibBlockInputSignalIsFrameData` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalLocalSampleTimeIndex (portIdx)

Returns the local sample time index corresponding to the specified block input port.

See `LibBlockInputSignalLocalSampleTimeIndex` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalNumDimensions(portIdx)

Returns the number of dimensions of the specified block input port.

See `LibBlockInputSignalNumDimensions` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalOffsetTime(portIdx)

Returns the offset time corresponding to the specified block input port.

See `LibBlockInputSignalOffsetTime` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalSampleTime(portIdx)

Returns the sample time corresponding to the specified block input port.

See `LibBlockInputSignalSampleTime` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalSampleTimeIndex(portIdx)

Returns the sample time index corresponding to the specified block input port.

See `LibBlockInputSignalSampleTimeIndex` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalWidth(portIdx)

Returns the width of the specified block input port index.

See `LibBlockInputSignalWidth` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockNumInputPorts(block)

Returns the number of data input ports of a block (excludes control ports).

See `LibBlockNumInputPorts` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

Output Signal Functions

In this section...

LibBlockAssignOutputSignal(portIdx, ucv, lcv, sigIdx, rhs)

Based on the output port number (`portIdx`), the user control variable (`ucv`), the loop control variable (`lcv`), the signal index (`sigIdx`), and the output signal destination, `LibBlockAssignOutputSignal` assigns a block's output to a specified right hand side value, (`rhs`).

See `LibBlockAssignOutputSignal` in matlabroot/rtw/c/tlc/mw/customstoragelib.tlc.

LibBlockNumOutputPorts(block)

Returns the number of data output ports of a block (excludes control and state ports).

See `LibBlockNumOutputPorts` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockOutputPortIndexMode(block, pidx)

Purpose

Determines the index mode of a block's output port.

Description

If a block's output port is set as an index port and its indexing base is marked as zero-based or one-based, this information is written into the *`model`*`.rtw` file. `LibBlockOutputPortIndexMode` queries the indexing base to branch to different code according to what the output port indexing base is.

Example

```
%if LibBlockOutputPortIndexMode(block, idx) == "Zero-based"
  ...
%elseif LibBlockOutputPortIndexMode(block, idx) == "One-based"
  ...
%else
  ...
%endif
```

Arguments

`block` — Block record

`pidx` — Port index

Returns

`""` for a nonindex port, and `"Zero-based"` or `"One-based"` otherwise.

See `LibBlockOutputPortIndexMode` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignal(portIdx, ucv, lcv, sigIdx)

Based on the output port number (`portIdx`), the user control variable (`ucv`), the loop control variable (`lcv`), the signal index (`sigIdx`), and the output signal destination, `LibBlockOutputSignal` returns a reference to a block output signal.

The returned value is a valid `lvalue` (left-side value) for an expression. The block output destination can be a location in the block I/O vector (another block's input), the state vector, or an external output.

Note Do not use `LibBlockOutputSignal` to access the address of an output signal.

The Simulink Coder code generator tracks when a variable (e.g., a signal or parameter) is accessed by its address. To access the address of an output signal, use `LibBlockOutputSignalAddr` as in the following example:

```
%assign yAddr = LibBlockOutputSignalAddr(0, "", lcv, sigIdx)
x = %<yAddr>;
```

See `LibBlockOutputSignal` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalAddr(portIdx, ucv, lcv, sigIdx)

Returns a string that provides the memory address of the specified block output port signal.

When an output signal address is required, you must use `LibBlockOutputSignalAddr` instead of taking the address that is returned by `LibBlockOutputSignal`. For example, `LibBlockOutputSignal` can return a literal constant, such as 5 (i.e., an invariant output signal). When `LibBlockOutputSignalAddr` is called on an invariant signal, the signal is declared as a `const` instead of being placed as a literal constant in the generated code.

Note that unlike `LibBlockOutputSignal`, the last argument, `sigIdx`, is not overloaded. Hence, if the output signal is complex, the address of the complex container is returned.

Example

To get the address of a wide output signal and pass it to a user function for processing, you could use

```
%assign u = LibBlockOutputSignalAddr(0, "", "", 0)
%assign y = LibBlockOutputSignal(0, "", "", 0)
%<y> = myfcn (%<u>);
```

See `LibBlockOutputSignalAddr` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalAliasedThruDataTypeName (portIdx, reim)

Returns the type name string (e.g., `int_T, ... creal_T`) of the aliased data type corresponding to the specified block output port.

Specify the `reim` argument as `""` if you want the complete signal type name. For example, if `reim == ""` and the first output port is real and complex, the data type placed in `dtname` is `creal_T`:

```
%assign dtname = LibBlockOutputSignalAliasedThroughDataTypeName(0x,"")
```

Specify the reim argument as `tRealPart` if you want the raw element type name. For example, if `reim == tRealPart` and the first output port is real and complex, the data type name returned is `real_T`.

```
%assign dtname = LibBlockOutputSignalAliasedThroughDataTypeName(0,tRealPart)
```

See `LibBlockOutputSignalAliasedThruDataTypeName` in `matlabroot/rtw/c/tlc/lib/blkiolib.tlc.`

LibBlockOutputSignalBeingMerged(portIdx)

Returns whether the specified output port is connected to a Merge block.

See `LibBlockOutputSignalBeingMerged` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalConnected(portIdx)

Returns 1 if the specified output port is connected to a block other than the Ground block and 0 otherwise.

See `LibBlockOutputSignalConnected` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalDataTypeId(portIdx)

Returns the numeric ID corresponding to the data type of the specified block output port.

If the output port signal is complex, `LibBlockOutputSignalDataTypeId` returns the data type of the real (or the imaginary) part of the signal.

See `LibBlockOutputSignalDataTypeId` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalDataTypeName(portIdx, reim)

Returns the type name string (e.g., `int_T`, ... `creal_T`) of the data type corresponding to the specified block output port.

Specify the `reim` argument as `""` if you want the complete signal type name. For example, if `reim==""` and the first output port is real and complex, the data type name placed in `dtname` is `creal_T`.

```
%assign dtname = LibBlockOutputSignalDataTypeName(0x,"")
```

Specify the `reim` argument as `tRealPart` if you want the raw element type name. For example, if `reim==tRealPart` and the first output port is real and complex, the data type name returned is `real_T`.

```
%assign dtname = LibBlockOutputSignalDataTypeName(0,tRealPart)
```

See `LibBlockOutputSignalDataTypeName` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalDimensions(portIdx)

Returns the dimensions of the specified block output port.

See `LibBlockOutputSignalDimensions` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalIsComplex(portIdx)

Returns 1 if the specified block output port is complex, 0 otherwise.

See `LibBlockOutputSignalIsComplex` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalIsFrameData(portIdx)

Returns 1 if the specified block output port is frame based, 0 otherwise.

See `LibBlockOutputSignalIsFrameData` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalLocalSampleTimeIndex (portIdx)

Returns the local sample time index corresponding to the specified block output port.

See `LibBlockOutputSignalLocalSampleTimeIndex` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalNumDimensions(portIdx)

Returns the number of dimensions of the specified block output port.

See `LibBlockOutputSignalNumDimensions` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalOffsetTime(portIdx)

Returns the offset time corresponding to the specified block output port.

See `LibBlockOutputSignalOffsetTime` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalSampleTime(portIdx)

Returns the sample time corresponding to the specified block output port.

See `LibBlockOutputSignalSampleTime` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalSampleTimeIndex(portIdx)

Returns the sample time index corresponding to the specified block output port.

See `LibBlockOutputSignalSampleTimeIndex` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalWidth(portIdx)

Returns the width of the specified block output port.

See `LibBlockOutputSignalWidth` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

Parameter Functions

In this section...

LibBlockMatrixParameter

`LibBlockMatrixParameter(param, rucv, rlcv, ridx, cucv, clcv, cidx)` returns a matrix parameter for a block, given the row and column user control variables (`rucv`, `cucv`), loop control variables (`rlcv`, `clcv`), and indices (`ridx`, `cidx`). Generally, blocks should use `LibBlockParameter`. If you have a matrix parameter, you should write it as a column-major vector and access it via `LibBlockParameter`.

Note Loop rolling is currently not supported, and will generate an error if requested (i.e., if either `rlcv` or `clcv` is not equal to `""`).

The row and column index arguments are similar to the arguments for `LibBlockParameter`. The column index (`cidx`) is overloaded to handle complex numbers.

See `LibBlockMatrixParameter` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockMatrixParameterAddr

`LibBlockMatrixParameterAddr(param, rucv, rlcv, ridx, cucv, clcv, cidx)` returns the address of a matrix parameter.

Note `LibBlockMatrixParameterAddr` returns the address of a matrix parameter. Loop rolling is not supported (i.e., `rlcv` and `clcv` should both be an empty string).

See `LibBlockMatrixParameterAddr` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockMatrixParameterBaseAddr

`LibBlockMatrixParameterBaseAddr(param)` returns the base address of a matrix parameter.

See `LibBlockMatrixParameterBaseAddr` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockParamSetting

`LibBlockParamSetting(bType, psType)` returns the string of a specified parameter setting for a specified block type. If you pass an empty block type into this function, the parameter setting will be assumed to be in the `ParamSettings` record of the block. If a nonempty block type is passed into the function, the parameter settings will be assumed to be in the `%<Btype>ParamSettings` record of that block.

See `LibBlockParamSetting` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockParameter

Based on the parameter reference (`param`), the user control variable (`ucv`), the loop control variable (`lcv`), the signal index (`sigIdx`), and the state of parameter inlining, `LibBlockParameter(param, ucv, lcv, sigIdx)` returns a reference to a block parameter. The returned value is a valid `rvalue` (right-side value for an expression). For example,

Case	Function Call	Can Produce
1	`LibBlockParameter(Gain, "i", lcv, sigIdx)`	`rtP.blockname[i]`
2	`LibBlockParameter(Gain, "i", lcv, sigIdx)`	`rtP.blockname`
3	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`p_Gain[i]`
4	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`p_Gain`
5	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`4.55`
6	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`rtP.blockname.re`
7	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`rtP.blockname.im`

To illustrate the basic workings of `LibBlockParameter`, assume a noncomplex vector signal where `Gain[0]=4.55`:

```
LibBlockParameter(Gain, "", "i", 0)
```

Case	Rolling	Inline Parameter	Type	Result	Required in Memory
1	0	Yes	Scalar	4.55	No
2	1	Yes	Scalar	4.55	No
3	0	Yes	Vector	4.55	No
4	1	Yes	Vector	`p_Gain[i]`	Yes
5	0	No	Scalar	`rtP.blk.Gain`	No
6	0	No	Scalar	`rtP.blk.Gain`	No
7	0	No	Vector	`rtP.blk.prm[0]`	No
8	0	No	Vector	`p.Gain[i]`	Yes

Note Case 4. Even though Inline Parameter is Yes, the parameter must be placed in memory (RAM), because it is accessed inside a `for` loop.

Note `LibBlockParameter` also supports expressions when used with inlined parameters and parameter tuning.

For example, if the parameter field had the MATLAB expression `'2*a'`, `LibBlockParameter` would return the C expression `'(2*a)'`. The list of

functions supported by `LibBlockParameter` is determined by the functions `FcnConvertNodeToExpr` and `FcnConvertIdToFcn`. To enhance functionality, augment or update either of these functions.

Note that certain types of expressions are not supported, such as `x*y` where *both* `x` and `y` are nonscalar expressions.

See the Simulink Coder documentation about tunable parameters for more details on the exact functions and syntax that are supported.

Warning

Do not use `LibBlockParameter` to access the address of a parameter, or you may might erroneously reference a number (i.e., `&4.55`) when the parameter is inlined. You can avoid this situation by using `LibBlockParameterAddr`.

See `LibBlockParameter` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockParameterAddr

`LibBlockParameterAddr(param, ucv, lcv, idx)` returns the address of a block parameter.

Using `LibBlockParameterAddr` to access a parameter when the global `InlineParameters` variable is equal to 1 will cause the variable to be declared `const` in RAM instead of being inlined.

Accessing the address of an expression when **Inline parameters** is set and the expression has multiple tunable/rolled variables in it will result in an error.

See `LibBlockParameterAddr` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockParameterBaseAddr

`LibBlockParameterBaseAddr(param)` returns the base address of a block parameter.

Using `LibBlockParameterBaseAddr` to access a parameter when the global `InlineParameters` variable is equal to one will cause the variable to be declared `const` in RAM instead of being inlined.

Accessing the address of an expression when **Inline parameters** is set and the expression has multiple tunable/rolled variables in it will result in an error.

See `LibBlockParameterBaseAddr` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockParameterDataTypeId

`LibBlockParameterDataTypeId(param)` returns the numeric ID corresponding to the data type of the specified block parameter.

See `LibBlockParameterDataTypeId` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockParameterDataTypeName

`LibBlockParameterDataTypeName(param, reim)` returns the name of the data type corresponding to the specified block parameter.

See `LibBlockParameterDataTypeName` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockParameterDimensions

`LibBlockParameterDimensions(param)` returns a row vector of length `N` (where `N >= 1`) giving the dimensions of the parameter data.

For example,

```
%assign dims  = LibBlockParameterDimensions("paramName")
%assign nDims = SIZE(dims,1)
%foreach i=nDims
   /* Dimension %<i+1> = %<dims[i]> */
%endforeach
```

`LibBlockParameterDimensions` differs from `LibBlockParameterSize` in that it returns the dimensions of the parameter data prior to collapsing the `Matrix` parameter to a column-major vector. The collapsing occurs for run-time parameters that have specified their `outputAsMatrix` field as `False`.

See `LibBlockParameterDimensions` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockParameterIsComplex

`LibBlockParameterIsComplex(param)` returns 1 if the specified block parameter is complex, 0 otherwise.

See `LibBlockParameterIsComplex` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockParameterSize

`LibBlockParameterSize(param)` returns a vector of size 2 in the format `[nRows, nCols]` where `nRows` is the number of rows and `nCols` is the number of columns.

See `LibBlockParameterSize` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockParameterString

`LibBlockParameterString(param)` returns the specified block parameter interpreted as a string, for example, this function returns:

- `STRINGOF(param.Value[0])` if the parameter is a row matrix
- `STRINGOF(param.Value)` otherwise

Note: It is an error to invoke this function with a matrix-valued parameter with more than one row.

If you are only accessing the parameter values using `LibBlockParameterString` or `LibBlockParameterValue`, you should consider converting the parameter to a `ParamSetting`. This produces more efficient code since the parameter is not declared as a variable in the code.

See `LibBlockParameterString` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockParameterValue

`LibBlockParameterValue(param, elIdx)` determines the numeric value of a parameter. If you are only accessing the parameter values using `LibBlockParameterValue` or `LibBlockParameterString`, you should consider converting the parameter to a `ParamSetting`. This produces more efficient code since the parameter is not declared as a variable in the code.

Example

If you want to generate code for a different integrator depending on a parameter for a block, you can use the following:

```
%assign mode = LibBlockParameterValue(Integrator, 0)
%switch (mode)
   %case 1
      %<CodeForIntegrator1>
      %break
   %case 2
      %<CodeForIntegrator2>
      %break
```

```
   %default
      Error: Unrecognized integrator value.
      %break
%endswitch
```

See `LibBlockParameterValue` in matlabroot/rtw/c/tlc/lib/paramlib.tlc.

LibBlockParameterWidth

`LibBlockParameterWidth(param)` returns the number of elements (width) of a parameter.

See `LibBlockParameterWidth` in `matlabroot/rtw/c/tlc/lib/paramlib.tlc.`

Block State and Work Vector Functions

In this section...

LibBlockAssignDWork(dwork, ucv, lcv, sigIdx, rhs)

Based on the block's dwork index or record (`dwork`), the user control variable (`ucv`), the loop control variable (`lcv`), and the signal index (`sigIdx`), `LibBlockAssignDWork` assigns a block's dwork to a specified right hand side value (`rhs`).

See `LibBlockAssignDWork` in matlabroot/rtw/c/tlc/mw/customstoragelib.tlc.

LibBlockContinuousState(ucv, lcv, idx)

Returns a string corresponding to the specified block continuous state (`CSTATE`) element.

See `LibBlockContinuousState` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockContinuousStateDerivative(ucv, lcv, idx)

Returns a string corresponding to the specified block continuous state (`CSTATE`) element.

See also `LibBlockDiscreteState`.

See `LibBlockContinuousStateDerivative` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockContStateDisabled(ucv, lcv, idx)

Returns a string corresponding to the specified block continuous state (`CSTATE`) element.

See also `LibBlockDiscreteState`.

See `LibBlockContStateDisabled` in `matlabroot/rtw/c/tlc/lib/blocklib.tlc`.

LibBlockDWork(dwork, ucv, lcv, idx)

Returns a string corresponding to the specified block `dwork` element. The last input argument is overloaded to handle complex `dworks`.

`idx = "re3"` — Returns the real part of element 3 if `dwork` is complex, otherwise returns element 3.

`idx = "im3"` — Returns the imaginary part of element 3 if `dwork` is complex, otherwise returns `""`.

`idx = "3"` — Returns the complex container of element 3 if `dwork` is complex, otherwise returns element 3.

If either `ucv` or `lcv` is specified (i.e., it is not equal to `""`) then the index part of the last input argument (`sigIdx`) is ignored.

See `LibBlockDWork` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockDWorkAddr(dwork, ucv, lcv, idx)

Returns a string corresponding to the address of the specified block `dwork` element.

See `LibBlockDWorkAddr` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockDWorkDataTypeId(dwork)

Returns the data type ID of the specified block `dwork`.

See `LibBlockDWorkDataTypeId` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockDWorkDataTypeName(dwork, reim)

Returns the data type name of the specified block `dwork`.

See `LibBlockDWorkDataTypeName` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockDWorkIsComplex(dwork)

Returns 1 if the specified block `dwork`is complex. Returns 0 otherwise.

See `LibBlockDWorkIsComplex` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockDWorkName(dwork)

Returns the name of the specified block `dwork`.

See `LibBlockDWorkName` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockDWorkStorageClass(dwork)

Returns the storage class of the specified block `dwork`.

See `LibBlockDWorkStorageClass` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockDWorkStorageTypeQualifier(dwork)

Returns the storage type qualifier of the specified block `dwork`.

See `LibBlockDWorkStorageTypeQualifier` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockDWorkUsedAsDiscreteState(dwork)

Returns 1 if the specified block `dwork` is used as a discrete state, returns 0 otherwise.

See `LibBlockDWorkUsedAsDiscreteState` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockDWorkWidth(dwork)

Returns the width of the specified block `dwork`.

See `LibBlockDWorkWidth` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockDiscreteState(ucv, lcv, idx)

Returns a string corresponding to the specified block discrete state (`DSTATE`) element.

See `LibBlockDiscreteState` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockIWork(definediwork, ucv, lcv, idx)

Returns a string corresponding to the specified block `IWORK` element. See `LibBlockRWork`.

See `LibBlockIWork` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockMode(ucv, lcv, idx)

Returns a string corresponding to the specified block `MODE` element.

See `LibBlockMode` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockNonSampledZC(ucv, lcv, NSZCIdx)

Returns a string corresponding to the specified block `NSZC`.

`LibBlockNonSampledZC` returns an element for the nonsampled zero-crossing state based on `ucv`, `lcv`, and `NSZCIdx`.

Arguments

`ucv` — User control variable string

`lcv` — Loop control variable string

`NSZCIdx` — Nonsampled zero-crossing index

See `LibBlockNonSampledZC` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockPWork(definedpwork, ucv, lcv, idx)

Returns a string corresponding to the specified block `PWORK` element. See `LibBlockRWork`.

See `LibBlockPWork` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockRWork(definedrwork, ucv, lcv, idx)

Returns a string corresponding to the specified block `RWORK` element. The first argument, `definedrwork`, is a symbol defined in the `mdlRTW` routine of the C MEX file with code like:

```
ssWriteRTWWorkVect([...], "RWork", [...], "MyRWorkName", [...])
```

Alternatively, if `RWork` defines have not been made, `definedrwork` is ignored and the raw `RWork` vector is accessed. In this case, uses in a loop rolling context are disallowed.

See `LibBlockRWork` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibBlockZCSignalValue(ucv, lcv, zcsIdx, zcElIdx)

Purpose

Returns a string corresponding to the specified block `ZCSignalValue`

Arguments

`ucv`

User control variable string.

`lcv`

Loop control variable string.

`zcsIdx`

zc signal Idx

`zcElIdx`

Idx of zc signal element in the zc signal

Description

`LibBlockZCSignalValue` returns an element for the zero crossing state based on `ucv`, `lcv`, and `zcsIdx`.

See `LibBlockZCSignalValue` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

Block Path and Error Reporting Functions

In this section...

LibBlockReportError(block, errorstring)

Use `LibBlockReportError` when reporting errors for a block. `LibBlockReportError` is designed to be used from block target files (e.g., the TLC file for an inlined S-function).

`LibBlockReportError` can be called with or without the block record scoped. To call the function without a block record scoped, pass the block record. To call the function when the block is scoped, pass `block = []`.

```
LibBlockReportError([],"error string")
  -- If block is scoped
LibBlockReportError(blockrecord,"error string")
  -- If block record is available
```

See `LibBlockReportError` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibBlockReportFatalError(block, errorstring)

Use `LibBlockReportFatalError` when reporting fatal (assert) errors for a block. Use `LibBlockReportFatalError` for defensive programming. Refer to "Generating Errors from TLC Files".

See `LibBlockReportFatalError` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibBlockReportWarning(block, warnstring)

Use `LibBlockReportWarning` when reporting warnings for a block. `LibBlockReportWarning` is designed to be used from block target files (e.g., the TLC file for an inlined S-function).

`LibBlockReportWarning` can be called with or without the block record scoped. To call this function without a block record scoped, pass the block record. To call this function when the block is scoped, pass `block = []`.

```
LibBlockReportWarning([],"warn string")
-- If block is scoped
LibBlockReportWarning(blockrecord,"warn string)
-- If block record is available
```

See `LibBlockReportWarning` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetBlockName(block)

`LibGetBlockName` returns the short block path name string for a block record, excluding carriage returns and other special characters that can be present in the name.

See `LibGetBlockName` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetBlockPath(block)

`LibGetBlockPath` returns the full block path name string for a block record, including carriage returns and other special characters that can be present in the name. Currently, the only other special string sequences defined are `'/*'` and `'*/'`.

The full block path name string is useful when you are accessing blocks from MATLAB. For example, you can use the full block name with `hilite_system` via `FEVAL` to match the Simulink path name exactly.

Use `LibGetFormattedBlockPath` to get a block path suitable for placing in a comment or error message.

See `LibGetBlockPath` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetFormattedBlockPath(block)

`LibGetFormattedBlockPath` returns the full path name string of a block without special characters. The string returned from `LibGetFormattedBlockPath` is suitable for placing the block name, in comments or generated code, on a single line.

Currently, the special characters are carriage returns, `'/*'`, and `'*/'`. A carriage return is converted to a space, `'/*'` is converted to `'/+'`, and `'*/'` is converted to `'+/'`. Note

that a '/' in the name is automatically converted to a '//' to distinguish it from a path separator.

Use `LibGetBlockPath` to get the block path for MATLAB functions used in reference blocks in your model.

See `LibGetFormattedBlockPath` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

Code Configuration Functions

In this section...

In this section...

LibAddSourceFileCustomSection (file, builtInSection, newSection)

Adds a custom section to a source file. You must associate a custom section with one of the built-in sections: Includes, Defines, Types, Enums, Definitions, Declarations, Functions, or Documentation. Nothing happens if the section already exists, except to report an error if a inconsistent built-in section association is attempted. `LibAddSourceFileCustomSection` is available only with the Embedded Coder product.

Arguments

`file` — Source file reference

`builtInSection` — Name of the associated built-in section

`newSection` — Name of the new (custom) section

See `LibAddSourceFileCustomSection` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibAddToCommonIncludes(incFileName)

Adds items to a list of `#include /package` specification items. Each member of the list is unique. Attempting to add a duplicate member does nothing.

`LibAddToCommonIncludes` should be called from block TLC methods to specify generation of `#include` statements in *`model`*`.h`. Specify the names of files on the include path inside angle brackets, e.g., `<sysinclude.h>`. Specify the names of local files without angle brackets, e.g., `myinclude.h`. Each call to `LibAddToCommonIncludes` adds the specified file to the list only if it is not already there. Filenames with and without angle brackets (e.g., `<math.h>` and `math.h`) are considered different. The `#include` statements are placed inside *`model`*`.h`.

Example

```
LibAddToCommonIncludes("tpu332lib.h")
```

See `LibAddToCommonIncludes` in matlabroot/rtw/c/tlc/lib/cachelib.tlc.

LibAddToModelSources(newFile)

`LibAddToModelSources` serves two purposes:

- To notify the Simulink Coder build process that it must build with the specified source file
- To update the `SOURCES: file1.c file2.c ...` comment in the generated code.

For inlined S-functions, `LibAddToModelSources` is generally called from `BlockTypeSetup`. `LibAddToModelSources` adds a filename to the list of sources for building this model. `LibAddToModelSources` returns `1` if the filename passed in was a duplicate (i.e., it was already in the sources list) and `0` if it was not a duplicate.

Use the `SFunctionModules` block parameter instead of `LibAddToModelSources` when writing S-functions. See Writing S-Functions.

See `LibAddToModelSources` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibCacheDefine(buffer)

Each call to `LibCacheDefine` appends your buffer to the existing cache buffer. For blocks, `LibCacheDefine` is generally called from `BlockTypeSetup`.

`LibCacheDefine` caches `#define` statements for inclusion in *`model`*`.h` (or *`model`*`_private.h`). Call `LibCacheDefine` from inside `BlockTypeSetup` to cache a `#define` statement. Each call to `LibCacheDefine` appends your buffer to the existing cache buffer. The `#define` statements are placed inside *`model`*`.h` (or *`model`*`_private.h`).

Example

```
%openfile buffer
#define INTERP(x,x1,x2,y1,y2) ( y1+((y2 - y1)/(x2 - x1))*(x-x1))
#define this that
%closefile buffer
%<LibCacheDefine(buffer)>
```

See `LibCacheDefine` in matlabroot/rtw/c/tlc/lib/cachelib.tlc.

LibCacheExtern(buffer)

`LibCacheExtern` should be called from inside `BlockTypeSetup` to cache an `extern` statement. Each call to `LibCacheExtern` appends your buffer to the existing cache buffer. The `extern` statements are placed in *`model`*`_private.h`.

Example

```
%openfile buffer
 extern real_T mydata;
%closefile buffer
%<LibCacheExtern(buffer)>
```

See `LibCacheExtern` in matlabroot/rtw/c/tlc/lib/cachelib.tlc.

LibCacheFunctionPrototype(buffer)

`LibCacheFunctionPrototype` should be called from inside `BlockTypeSetup` to cache a function prototype. Each call to `LibCacheFunctionPrototype` appends your buffer to the existing cache buffer. The prototypes are placed inside *`model`*`_private.h`.

Example

```
%openfile buffer
 extern int_T fun1(real_T x);
 extern real_T fun2(real_T y, int_T i);
%closefile buffer
%<LibCacheFunctionPrototype(buffer)>
```

See `LibCacheFunctionPrototype` in matlabroot/rtw/c/tlc/lib/cachelib.tlc.

LibCacheTypedefs(buffer)

`LibCacheTypedefs` should be called from inside `BlockTypeSetup` to cache `typedef` declarations. Each call to `LibCacheTypedefs` appends your buffer to the existing cache buffer. The `typedef` statements are placed inside *`model`*`.h` (or *`model`*`_common.h`).

Example

```
%openfile buffer
typedef foo bar;
%closefile buffer
%<LibCacheTypedefs(buffer)>
```

See `LibCacheTypedefs` in matlabroot/rtw/c/tlc/lib/cachelib.tlc.

LibCallModelInitialize()

Returns code for calling the model's initialize function (valid for ERT only).

See `LibCallModelInitialize` in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc`.

LibCallModelStep(tid)

Returns code for calling the model's step function (valid for ERT only).

See `LibCallModelStep` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibCallModelTerminate()

Returns code for calling the model's terminate function (valid for ERT only).

See `LibCallModelTerminate` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibCallSetEventForThisBaseStep(buffername)

Returns code for calling the model's set events function (valid for ERT only).

Argument

`buffername` — Name of the variable used to buffer the events. For the example `ert_main.c`, this is `eventFlags`.

See `LibCallSetEventForThisBaseStep` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibCreateSourceFile(type, creator, name)

Creates a new C or C++ file and returns its reference. If the file already exists, `LibCreateSourceFile` returns the existing file's reference.

Syntax

```
%assign fileH = LibCreateSourceFile
                ("Source", "Custom", "foofile")
```

Arguments

`type` (string) — Valid values are `"Source"` and `"Header"` for `.c` and `.h` files, respectively.

`creator` (string) — Who is creating the file? An error is reported if different creators attempt to create the same file.

`name` (string) — Base name of the file (i.e., without the extension). Note that files are not written to disk if they are empty.

Returns

Reference to the model file (scope).

See `LibCreateSourceFile` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibGetFileRecordName (file)

Returns model file name (including the path) without the file extension. To retrieve the file name (including the path) with the file extension, use `LibGetSourceFileSection`.

Arguments

`file` — Source file reference

See `LibGetFileRecordName` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibGetMdlPrvHdrBaseName()

Returns the base name of the model's private header file, for example, *model*_private.h.

See LibGetMdlPrvHdrBaseName in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibGetMdlPubHdrBaseName()

Returns the base name of the model's public header file, for example, *model*.h.

See LibGetMdlPubHdrBaseName in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibGetMdlSrcBaseName()

Returns the base name of the model's main source file, for example, *model*.c.

See LibGetMdlSrcBaseName in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibGetModelDotCFile()

Returns a reference to the *model*.c or .cpp source file. You can then cache additional code using LibSetSourceFileSection.

Syntax

```
%assign srcFile = LibGetModelDotCFile()
%<LibSetSourceFileSection(srcFile, "Functions", mybuf)>
```

Returns

Returns a reference to the *model*.c or .cpp source file.

See LibGetModelDotCFile in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibGetModelDotHFile()

Returns a reference to the *model*.h source file. You can then cache additional code using LibSetSourceFileSection.

Syntax

```
%assign hdrFile = LibGetModelDotHFile()
%<LibSetSourceFileSection(hdrFile, "Functions", mybuf)>
```

Returns

Returns a reference to the *`model`*`.h` source file.

See `LibGetModelDotHFile` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibGetModelName()

Returns the name of the model (without an extension).

See `LibGetModelName` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibGetNumSourceFiles()

Returns the number of source files (`.c` or `.cpp` and `.h`) that have been created.

Syntax

```
%assign numFiles = LibGetNumSourceFiles()
```

Returns

Returns the number of files (number).

See `LibGetNumSourceFiles` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibGetRTModelErrorStatus()

Returns the code required to get the model error status.

Syntax

```
%<LibGetRTModelErrorStatus()>;
```

See `LibGetRTModelErrorStatus` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibGetSourceFileCustomSection(file, attrib)

Gets a custom section previously created with `LibAddSourceFileCustomSection`.

Arguments

`file` (scope or number) — Source file reference or index

`attrib` (string) — Name of custom section

See `LibGetSourceFileCustomSection` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibGetSourceFileFromIdx(fileIdx)

Returns a model file reference based on its index. This reference can be useful for a common operation on all files, for example, to set the leading file banner of all files.

Syntax

```
%assign fileH = LibGetSourceFileFromIdx(fileIdx)
```

Argument

`fileIdx` (number) — Index of model file

Returns

Reference (scope) to the model file.

See `LibGetSourceFileFromIdx` in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc`.

LibGetSourceFileTag(fileIdx)

Returns *`fileName`*`_h` and *`fileName`*`_c` for header and source files, respectively, where *`fileName`* is the name of the model file.

Syntax

```
%assign tag = LibGetSourceFileTag(fileIdx)
```

Argument

`fileIndex` (number) — File index

Returns

Returns the tag (string).

See `LibGetSourceFileTag` in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc`.

LibMdlRegCustomCode(buffer, location)

Places declaration statements and executable code inside the *`model`*`_initialize` function.

Arguments

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

See `LibMdlRegCustomCode` in matlabroot/rtw/c/tlc/mw/hookslib.tlc.

LibMdlStartCustomCode(buffer, location)

Places declaration statements and executable code inside the start function. Start code is executed once, during the model initialization phase.

Syntax

```
LibMdlStartCustomCode(buffer, location)
```

Arguments

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibMdlStartCustomCode` places declaration statements and executable code inside the start function. This code is output into the following functions, depending on the current code format:

Function Name	Code Format
`model``_initialize`	Embedded-C
`mdlStart`	S-function
`MdlStart`	RealTime

Each call to `LibMdlStartCustomCode` appends your buffer to the internal cache buffer.

See `LibMdlStartCustomCode` in matlabroot/rtw/c/tlc/mw/hookslib.tlc.

LibMdlTerminateCustomCode(buffer, location)

Purpose

Places declaration statements and executable code inside the terminate function.

Syntax

```
LibMdlTerminateCustomCode(buffer, location)
```

Arguments

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibMdlTerminateCustomCode` places declaration statements and executable code inside the terminate function. This code is output into the following functions, depending on the current code format:

Function Name	Code Format
`model``_terminate`	Embedded-C
`mdlTerminate`	S-function
`MdlTerminate`	RealTime

Each call to `LibMdlTerminateCustomCode` appends your buffer to the internal cache buffer.

See `LibMdlTerminateCustomCode` in matlabroot/rtw/c/tlc/mw/hookslib.tlc.

LibSetRTModelErrorStatus(str)

Returns the code required to set the model error status.

Syntax

```
LibSetRTModelErrorStatus("\"Overrun\"")
```

Argument

`str` (string) — char * to a C string

See `LibSetRTModelErrorStatus` in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc`.

LibSetSourceFileCodeTemplate(opFile, name)

By default, `*.c` and `*.h` files are generated with the code templates specified in the **Code Generation > Templates** pane of the Configuration Parameters dialog box. `LibSetSourceFileCodeTemplate` allows you to change the template for a file.

Note: Custom templates are a feature of the Embedded Coder product.

Syntax

```
%assign tag = LibSetSourceFileCodeTemplate(opFile,name)
```

Arguments

`opFile` (scope) — Reference to file

`name` (string) — Name of the desired template

Returns

Nothing

See `LibSetSourceFileCodeTemplate` in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc`.

LibSetSourceFileCustomSection(file, attrib, value)

Adds to the contents of a custom section previously created with `LibAddSourceFileCustomSection`. Available only with Embedded Coder software.

Arguments

`file` (scope or number) — Source file reference or index

`attrib` (string) — Name of custom section

`value` (string) — Value to be appended to section

See `LibSetSourceFileCustomSection` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibSetSourceFileOutputDirectory(opFile, name)

By default, `*.c` and `*.h` files are generated into a Simulink Coder build folder at the current location. `LibSetSourceFileOutputDirectory` allows you to change the folder into which a specified source file is to be generated. Note that the caller is responsible for specifying a valid folder.

Syntax

```
%assign tag = LibSetSourceFileOutputDirectory(opFile,dirName)
```

Arguments

`opFile` (scope) — Reference to file

`dirName` (string) — Name of the desired output folder

Returns

Nothing

See `LibSetSourceFileOutputDirectory` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibSetSourceFileSection(fileH, section, value)

Adds to the contents of a specified section within a specified file. Valid file sections include

File Section	Description
`Banner`	Set the file banner (comment) at the top of the file.
`Includes`	Append to the `#include` section.
`Defines`	Append to the `#define` section.
`IntrinsicTypes`	Append to the intrinsic `typedef` section. Intrinsic types are those that depend only on intrinsic C types.
`PrimitiveTypedefs`	Append to the primitive `typedef` section. Primitive `typedef`s are those that depend only on intrinsic C types and `typedef`s previously defined in the `IntrinsicTypes` section.
`UserTop`	Append to the User Top section.
`Typedefs`	Append to the `typedef` section. The `typedef`s can depend on a previously defined type.
`Enums`	Append to the enumerated types section.
`Definitions`	Append to the data definition section.
`ExternData`	(Reserved) Simulink Coder `extern` data.
`ExternFcns`	(Reserved) Simulink Coder `extern` functions.
`FcnPrototypes`	(Reserved) Simulink Coder function prototypes.
`Declarations`	Append to the data declaration section.
`Functions`	Append to the C functions section.
`CompilerErrors`	Append to the `#error` section.
`CompilerWarnings`	Append to the `#warning` section.
`Documentation`	Append to the documentation (comment) section.
`UserBottom`	Append to the User Bottom section.

The code generator orders the code as listed above.

Syntax

Example (iterating over the files):

```
%openfile tmpBuf
  whatever
%closefile tmpBuf

%foreach fileIdx = LibGetNumSourceFiles()
  %assign fileH = LibGetSourceFileFromIdx(fileIdx)
  %<LibSetSourceFileSection(fileH,"SectionOfInterest",tmpBuf)>
%endforeach

%assign fileH = LibCreateSourceFile("Header","Custom","foofile")
%<LibSetSourceFileSection(fileH,"Defines","#define FOO 5.0\n")>
```

Arguments

`fileH` (scope or number) — Reference or index to a file

`section` (string) — File section of interest

`value` (string) — Value

See `LibSetSourceFileSection` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibSystemDerivativeCustomCode (system, buffer, location)

Purpose

Places declaration statements and executable code inside a subsystem's derivative function.

Syntax

```
LibSystemDerivativeCustomCode(system, buffer, location)
```

Arguments

`system` — Reference to the subsystem whose derivative function is to be modified.

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function

- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibSystemDerivativeCustomCode` places declaration statements and executable code inside the derivative function for the subsystem specified by `system`. This code is output into the following functions, depending on the current code format:

Function Name	Code Format
`mdlDerivatives`	S-function
`model``_derivatives`	RealTime

`LibSystemDerivativeCustomCode` is not relevant for the Embedded-C code format, because blocks with continuous states cannot be used.

Each call to `LibSystemDerivativeCustomCode` appends your buffer to the internal cache buffer. An error is generated if you attempt to add code to a subsystem that does not have continuous states.

See `LibSystemDerivativeCustomCode` in matlabroot/rtw/c/tlc/mw/hookslib.tlc.

LibSystemDisableCustomCode (system, buffer, location)

Purpose

Places declaration statements and executable code inside a subsystem's disable function.

Syntax

```
LibSystemDisableCustomCode(system, buffer, location)
```

Arguments

`system` — Reference to the subsystem whose disable function is to be modified.

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibSystemDisableCustomCode` places declaration statements and executable code inside the disable function for the subsystem specified by `system`. Each call to `LibSystemDisableCustomCode` appends your buffer to the internal cache buffer.

An error is generated if you attempt to add code to a subsystem that does not have a disable function.

See `LibSystemDisableCustomCode` in matlabroot/rtw/c/tlc/mw/hookslib.tlc.

LibSystemEnableCustomCode (system, buffer, location)

Purpose

Places declaration statements and executable code inside a subsystem's enable function.

Syntax

```
LibSystemEnableCustomCode(system, buffer, location)
```

Arguments

`system` — Reference to the subsystem whose enable function is to be modified.

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibSystemEnableCustomCode` places declaration statements and executable code inside the enable function for the subsystem specified by `system`. Each call to `LibSystemEnableCustomCode` appends your buffer to the internal cache buffer.

An error is generated if you attempt to add code to a subsystem that does not have an enable function.

See `LibSystemEnableCustomCode` in matlabroot/rtw/c/tlc/mw/hookslib.tlc.

LibSystemInitializeCustomCode (system, buffer, location)

Purpose

Places declaration statements and executable code inside a subsystem's initialize function.

Syntax

```
LibSystemInitializeCustomCode(system, buffer, location)
```

Arguments

`system` — Reference to the subsystem whose initialize function is to be modified.

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibSystemInitializeCustomCode` places declaration statements and executable code inside the initialize function for the subsystem specified by `system`. This code is output into the following functions, depending on the current code format:

Function Name	Code Format
`model``_initialize`	Embedded-C
`mdlInitializeConditions`	S-function
`MdlStart`	RealTime

Code for a subsystem is output into the subsystem's initialization function. Each call to `LibSystemInitializeCustomCode` appends your buffer to the internal cache buffer.

Note Enable systems that are not configured to reset on enable are inlined into `MdlStart`. For this case, the subsystem's custom code is found in `MdlStart` above and below the enable subsystem's initialization code.

See `LibSystemInitializeCustomCode` in matlabroot/rtw/c/tlc/mw/hookslib.tlc.

LibSystemOutputCustomCode (system, buffer, location)

Purpose

Places declaration statements and executable code inside a subsystem's output function.

Syntax

```
LibSystemOutputCustomCode(system, buffer, location)
```

Arguments

`system` — Reference to the subsystem whose output function is to be modified.

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibSystemOutputCustomCode` places declaration statements and executable code inside the output function for the subsystem specified by `system`. This code is output into the following functions, depending on the current code format:

Function Name	Code Format
`model``_step`	Embedded-C (`CombineOutputUpdateFcns` is `1`)
`model``_output`	Embedded-C (`CombineOutputUpdateFcns` is `0`)
`mdlOutputs`	S-function
`MdlOutputs`	RealTime

Each call to `LibSystemOutputCustomCode` appends your buffer to the internal cache buffer.

See `LibSystemOutputCustomCode` in `matlabroot/rtw/c/tlc/mw/hookslib.tlc`.

LibSystemUpdateCustomCode

(system, buffer, location)

Purpose

Places code inside a subsystem's update function.

Syntax

```
LibSystemUpdateCustomCode(system, buffer, location)
```

Arguments

`system` — Reference to the subsystem whose update function is to be modified.

`buffer` — String buffer containing text to append to the internal cache buffer.

`location` — String specifying where to place the buffer's contents. Possible values are

- `"header"` — Place buffer at top of function
- `"declaration"` — Place buffer at top of function
- `"execution"` — Place buffer at top of function, but after header
- `"trailer"` — Place buffer at bottom of function

Returns

Nothing

Description

`LibSystemUpdateCustomCode` places declaration statements and executable code inside the update function for the subsystem specified by `system`. This code is output into the following functions, depending on the current code format:

Function Name	Code Format
`model``_step`	Embedded-C (`CombineOutputUpdateFcns` is `1`)
`model``_update`	Embedded-C (`CombineOutputUpdateFcns` is `0`)
`mdlUpdate`	S-function
`MdlUpdate`	RealTime

Each call to `LibSystemUpdateCustomCode` appends your buffer to the internal cache buffer.

See `LibSystemUpdateCustomCode` in matlabroot/rtw/c/tlc/mw/hookslib.tlc.

LibWriteModelData()

Returns data for the model (valid for ERT only).

See `LibWriteModelData` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibWriteModelInput(tid, rollThreshold)

Returns the code for writing to a specified root input (that is, a model inport block). This function is valid for ERT only, and not valid for referenced models.

Arguments

`tid` (number) — Task identifier (`0` is fastest rate and `n` is the slowest)

`rollThreshold` — Width of signal before wrapping in a `for` loop.

See `LibWriteModelInput` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibWriteModelInputs()

Returns code that writes to all root inputs (that is, the model inport blocks). This function is valid for ERT only, and is not valid for referenced models.

See `LibWriteModelInputs` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibWriteModelOutput(tid, rollThreshold)

Returns code that writes to a specified root output (that is, a model outport block). This function is valid for ERT only, and not valid for referenced models.

Arguments

`tid` (number) — Task identifier (`0` is fastest rate and `n` is the slowest)

`rollThreshold` — Width of signal before wrapping in a `for` loop.

See `LibWriteModelOutput` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibWriteModelOutputs()

Returns code that writes to all root outputs (that is, the model outport blocks). This function is valid for ERT only, and not valid for referenced models.

See `LibWriteModelOutputs` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

Sample Time Functions

In this section...

In this section...

LibAsynchronousTriggeredTID(tid)

Returns whether this TID corresponds to a asynchronous triggered rate.

See `LibAsynchronousTriggeredTID` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibAsyncTaskAccessTimeInFcn(tid, fcnType)

Returns 1 if the specified asynchronous task identifier (TID) is given function type.

See `LibAsyncTaskAccessTimeInFcn` in matlabroot/rtw/c/tlc/mw/sutillib.tlc.

LibBlockSampleTime(block)

Returns the block's sample time. The returned value depends on the sample time classification of the block, as shown in the following table.

Block Classification	Returned Value
Discrete	The actual sample time of a block (real > 0)
Continuous	`0.0`
Triggered	`-1.0`
Constant	`-2.0`

See `LibBlockSampleTime` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibGetClockTick(tid)

Returns integer task time (current clock tick of the task timer). The resolution of the timer can be obtained from `LibGetClockTickStepSize(tid)`. The data type ID of the timer can be obtained from `LibGetClockTickDataTypeId(tid)`.

See `LibGetClockTick` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetClockTickDataTypeId(tid)

Returns clock tick data type ID.

See `LibGetClockTickDataTypeId` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetClockTickHigh(tid)

Returns high-order word of the integer task time. `LibGetClockTickHigh` is used when `uint32` pairs are used to store absolute time. The resolution of a clock tick can be obtained from `LibGetClockTickStepSize(tid)`.

See `LibGetClockTickHigh` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetClockTickStepSize(tid)

Returns clock tick step size, which is the resolution of the integer task time. `LibGetClockTickStepSize` cannot be used if the task does not have a timer.

See `LibGetClockTickStepSize` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetElapseTime(system)

Returns time elapsed since the last time the subsystem started to execute.

See `LibGetElapseTime` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetElapseTimeCounter(system)

Returns an integer elapsed time. This is the number of clock ticks elapsed since the last time the system started. To get real-world elapsed time, this integer elapsed time must be multiplied by the applicable resolution.

You can obtain the resolution by calling `LibGetElapseTimeResolution(system)`. You can obtain the data type ID of the integer elapsed time counter by calling `LibGetElapseTimeCounterDTypeId(system)`.

See `LibGetElapseTimeCounter` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetElapseTimeCounterDTypeId(system)

Returns the date type ID of the integer elapsed time returned by `LibGetElapseTimeCounter`.

See `LibGetElapseTimeCounterDTypeId` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetElapseTimeResolution(system)

Returns the resolution of the elapsed time returned by `LibGetElapseTimeCounter`.

See `LibGetElapseTimeResolution` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetGlobalTIDFromLocalSFcnTID(sfcnTID)

Returns the model task identifier (sample time index) corresponding to the specified local S-function task identifier or port sample time. `LibGetGlobalTIDFromLocalSFcnTID` allows you to use one function to determine a global TID, independent of port- or block-based sample times.

The input argument to `LibGetGlobalTIDFromLocalSFcnTID` should be one of the following:

- For block-based sample times (e.g., in S-function `mdlInitializeSizes`, `ssSetNumSampleTimes(S, N)` with `N > 1` was specified), `sfcnTID` is a nonnegative integer giving the corresponding local S-function sample time.
- For port-based sample times (e.g., in S-function `mdlInitializeSizes`, `ssSetNumSampleTimes(S, PORT_BASED_SAMPLE_TIMES)` was specified), `sfcnTID` is a string of the form `"InputPortIdx`*I*`"` or `"OutputPortIdx`*I*`"`, where *I* is an integer ranging from 0 to the number of ports (e.g., `"InputPortIdx0"`.

Examples

Multirate block

```
%assign globalTID = LibGetGlobalTIDFromLocalSFcnTID(2)
```

or

```
%assign globalTID =
LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx4")
%assign period =
CompiledModel.SampleTime[globalTID].PeriodAndOffset[0]
%assign offset =
CompiledModel.SampleTime[globalTID].PeriodAndOffset[1]
```

Inherited sample time block

```
%switch (LibGetSFcnTIDType(0))
  %case "discrete"
  %case "continuous"
    %assign globalTID = LibGetGlobalTIDFromLocalSFcnTID(2)
    %assign period = ...
      CompiledModel.SampleTime[globalTID].PeriodAndOffset[0]
    %assign offset = ...
      CompiledModel.SampleTime[globalTID].PeriodAndOffset[1]
    %breaksw
  %case "triggered"
    %assign period = -1
    %assign offset = -1
    %breaksw
  %case "constant"
    %assign period = rtInf
    %assign offset = 0
    %breaksw
  %default
    %<LibBlockReportFatalError([],"Unknown tid type")>
%endswitch
```

See `LibGetGlobalTIDFromLocalSFcnTID` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetNumAsyncTasks()

Return the number of asynchronous tasks in generated code.

See `LibGetNumAsyncTasks` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetNumSFcnSampleTimes(block)

Returns the number of S-function sample times for a block.

See `LibGetNumSFcnSampleTimes` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetNumSyncPeriodicTasks()

Return the number of periodic tasks in generated code.

See `LibGetNumSyncPeriodicTasks` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetNumTasks()

Return the number of tasks in generated code.

See `LibGetNumTasks` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetSampleTimePeriodAndOffset(tid, idx)

Returns the sample time period value or offset value for a specified task.

Arguments

`tid` — Specify the identifier of the task for which to return information.

`idx` — Specify 0 to return the sample time period value or 1 to return the sample time offset value.

Examples

```
%% Get sample time period and offset for task 0
%assign sampleTime = LibGetSampleTimePeriodAndOffset(0,0)
%assign offsetTime = LibGetSampleTimePeriodAndOffset(0,1)

%% Get sample time periods for tasks 0 and 1
%assign periodTID0 = LibGetSampleTimePeriodAndOffset(0,0)
%assign periodTID1 = LibGetSampleTimePeriodAndOffset(1,0)
```

See `LibGetSampleTimePeriodAndOffset` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibGetSFcnTIDType(sfcnTID)

Returns the type of the specified S-function task identifier (`sfcnTID`). Possible values are:

- `"continuous"` — The specified `sfcnTID` is continuous.
- `"discrete"` — The specified `sfcnTID` is discrete.
- `"triggered"` — The specified `sfcnTID` is triggered.
- `"constant"` — The specified `sfcnTID` is constant.

The format of `sfcnTID` must be the same as for `LibIsSFcnSampleHit`.

Note This is useful primarily in the context of S-functions that specify an inherited sample time.

See `LibGetSFcnTIDType` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetTaskTime(tid)

Returns a string to access the absolute time of the task, which is a floating-point number. However, if you have configured the model for integer-only code generation (by deselecting **Support floating-point numbers**), the string represents an integer equal to the number of base rate ticks (the absolute time divided by the base sample time) rather than the absolute time. In integer-only code, the task time is an integer time value whose resolution is the model fundamental step size.

`LibGetTaskTime` is the TLC version of the SimStruct macro: `"ssGetTaskTime(S,tid)"`.

See `LibGetTaskTime` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetTaskTimeFromTID(block)

Returns a string to access the absolute time of the task associated with the block.

If the code format is not Embedded-C, `LibGetTaskTimeFromTID` returns the string `"RTMGet("T")"` if the block is constant or the system is single rate, and `"RTMGetTaskTimeForTID(tid)"` otherwise.

If the code format is Embedded-C, `LibGetTaskTimeFromTID` returns `"RTMGetTaskTimeForTID(tid)"`.

If the block is constant or the system is single rate, this is the TLC version of the SimStruct macro: `"ssGetT(S)"` and `"ssGetTaskTime(S, tid)"` otherwise.

In both cases, `S` is the name of the SimStruct.

See `LibGetTaskTimeFromTID` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetTID01EQ()

Returns the value of the TID01EQ flag — true (1) if sampling rates of the continuous task and the first discrete task are equal and false (0) otherwise.

When a model has continuous time and the flag TID01EQ is true, both continuous time and the fastest discrete time are treated as one rate in generated code. The code associated with the fastest discrete rate is guarded by a major time step check. When the model has only two rates, and TID01EQ is true, the generated code has a single-rate call interface.

Use `LibGetTID01EQ` to detect and account for the case where continuous time and the fastest discrete time are treated as one rate.

See `LibGetTID01EQ` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc, bareboard_mrmain.tlc, and ertmainlib.tlc.

LibIsContinuous(TID)

Returns 1 if the specified task identifier (`TID`) is continuous and 0 otherwise. Note that task identifiers equal to `"triggered"` or `"constant"` are not continuous.

See `LibIsContinuous` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibIsDiscrete(TID)

Returns 1 if the specified task identifier (`TID`) is discrete and 0 otherwise. Note that task identifiers equal to `"triggered"` or `"constant"` are not discrete.

See `LibIsDiscrete` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibIsSFcnSampleHit(sfcnTID)

Returns 1 if a sample hit occurs for the specified local S-function task identifier (`TID`) and 0 otherwise.

The input argument to `LibIsSFcnSampleHit` should be one of the following:

- `sfcnTID`: integer (e.g., 2)

  For block-based sample times (e.g., in S-function `mdlInitializeSizes`, `ssSetNumSampleTimes(S,N)` with N > 1 was specified), `sfcnTID` is an integer starting at 0 of the corresponding local S-function sample time.
- `sfcnTID`: `"InputPortIdx`*`I`*`"` or `"OutputPortIdx`*`I`*`"` (e.g., `"InputPortIdx0"`).

  For port-based sample times (e.g., in S-function `mdlInitializeSizes`, `ssSetNumSampleTimes(S,PORT_BASED_SAMPLE_TIMES)` was specified), `sfcnTID` is a string giving the input (or output) port index.

Examples

- Consider a multirate S-function block with four block sample times. The call `LibIsSFcnSampleHit(2)` returns the code to check for a sample hit on the third S-function block sample time.
- Consider a multirate S-function block with three input and eight output sample times. The call `LibIsSFcnSampleHit("InputPortIdx0")` returns the code to check for a sample hit on the first input port. The call `LibIsSFcnSampleHit("OutputPortIdx7")` returns the code to check for a sample hit on the eighth output port.

See `LibIsSFcnSampleHit` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibIsSFcnSingleRate(block)

Returns a Boolean value (1 or 0) indicating whether the S-function is single rate (one sample time) or multirate (multiple sample times).

See `LibIsSFcnSingleRate` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibIsSFcnSpecialSampleHit(sfcnSTI, sfcnTID)

Returns the Simulink macro to promote a slow task (`sfcnSTI`) into a faster task (`sfcnTID`).

This advanced function is specifically intended for use in rate transition blocks. `LibIsSFcnSpecialSampleHit` determines the global TID from the S-function TID

The input arguments to `LibIsSFcnSpecialSampleHit` are

- For multirate S-function blocks:

  `sfcnSTI`: local S-function sample time index (`sti`) of the slow task that is to be promoted

  `sfcnTID`: local S-function task ID (`tid`) of the fast task where the slow task is to run
- For single-rate S-function blocks using `SS_OPTION_RATE_TRANSITION`, `sfcnSTI` and `sfcnTID` are ignored and should be specified as `""`.

The format of `sfcnSTI` and `sfcnTID` must follow that of the argument to `LibIsSFcnSampleHit`.

Examples

- A rate transition S-function (one sample time with `SS_OPTION_RATE_TRANSITION`)

  ```
  if (%<LibIsSFcnSpecialSampleHit("","")>) {
  ```
- A multirate S-function with port-based sample times where the output rate is slower than the input rate (e.g., a zero-order hold operation)

  ```
  if (%<LibIsSFcnSpecialSampleHit("OutputPortIdx0","InputPortIdx0")>) {
  ```

See `LibIsSFcnSpecialSampleHit` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibIsSingleRateModel()

Returns `true` if model is single rate and `false` otherwise.

See `LibIsSingleRateModel` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibIsSingleTasking()

Returns `true` if the model is configured for single-tasking execution and `false` if the model is configured for multitasking execution.

See `LibIsSingleTasking` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibIsZOHContinuous(TID)

Returns 1 if the specified task identifier (TID) is zero order hold (ZOH) continuous and 0 otherwise. A TID equal to `triggered` or `constant` is not ZOH continuous.

See `LibIsZOHContinuous` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibNumAsynchronousSampleTimes()

Returns the number of asynchronous sample times in the model.

See `LibNumAsynchronousSampleTimes` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibNumDiscreteSampleTimes()

Returns the number of discrete sample times in the model.

See `LibNumDiscreteSampleTimes` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibNumSynchronousSampleTimes()

Returns the number of synchronous sample times in the model.

See `LibNumSynchronousSampleTimes` in matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc.

LibPortBasedSampleTimeBlockIsTriggered(block)

Determines whether the port-based S-function block is triggered.

See `LibPortBasedSampleTimeBlockIsTriggered` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibSetVarNextHitTime(block, tNext)

Generates code to set the next variable hit time. Blocks with variable sample time must call `LibSetVarNextHitTime` in their output functions.

See `LibSetVarNextHitTime` in matlabroot/rtw/c/tlc/lib/blocklib.tlc.

LibTriggeredTID(tid)

Returns whether this TID corresponds to a triggered rate.

See `LibTriggeredTID` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

Miscellaneous Functions

In this section...

In this section...

LibBlockExecuteFcnCall(block, callIdx)

For use by inlined S-functions with function-call outputs. Returns a string to call the function-call subsystem with required arguments or generates the subsystem's code in place (inlined).

Example

```
%foreach callIdx = NumSFcnSysOutputCalls
  %if LibIsEqual(SFcnSystemOutputCall[callIdx].BlockToCall,...
   "unconnected")
    %continue
  %endif
%% call the downstream system
  %<LibBlockExecuteFcnCall(block, callIdx)>\
%endforeach
```

See `LibBlockExecuteFcnCall` in matlabroot/rtw/c/tlc/lib/syslib.tlc.

LibBlockExecuteFcnDisable(block, callIdx)

For use by inlined S-Functions to call the function-call system disable function. Returns a string to either call the function-call subsystem disable function with required arguments or to call the generated subsystem disable code (inlined).

Example

```
%foreach callIdx = NumSFcnSysOutputCallDsts
```

```
  %if LibIsEqual(SFcnSystemOutputCall[callIdx].BlockToCall, "unconnected")
   %continue
  %endif
%% call the downstream system
  %<LibBlockExecuteFcnDisable(block, callIdx)>\
%endforeach
```

See `LibBlockExecuteFcnDisable` in matlabroot/rtw/c/tlc/lib/syslib.tlc.

LibBlockExecuteFcnEnable(block, callIdx)

For use by inlined S-Functions to call the function-call system enable function. Returns a string to call the function-call subsystem enable function with required arguments or the generated subsystem enable code (inlined).

Example

```
%foreach callIdx = NumSFcnSysOutputCallDsts
  %if LibIsEqual(SFcnSystemOutputCall[callIdx].BlockToCall, "unconnected")
   %continue
  %endif
%% call the downstream system
  %<LibBlockExecuteFcnEnable(block, callIdx)>\
%endforeach
```

See `LibBlockExecuteFcnEnable` in matlabroot/rtw/c/tlc/lib/syslib.tlc.

LibBlockInputSignalAliasedThruDataTypeId(idx)

Return the data type ID the input signal is aliased thru to.

See LibBlockInputSignalAliasedThruDataTypeId in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibBlockOutputSignalAliasedThruDataTypeId(idx)

Return the data type ID the output signal is aliased thru to.

See `LibBlockOutputSignalAliasedThruDataTypeId` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc

LibGenConstVectWithInit(data, typeId, varId)

Returns an initialized static constant variable string of form:

```
static const typeName varId[] = { data };
```

The `typeName` is generated from `typeId`, which can be one of

```
tSS_DOUBLE, tSS_SINGLE, tSS_BOOLEAN, tSS_INT8, tSS_UINT8,
tSS_INT16, tSS_UINT16, tSS_INT32, tSS_UINT32
```

The data input argument must be a numeric scalar or vector and must be finite (no `Inf`, `-Inf`, or `NaN` values).

See `LibGenConstVectWithInit` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibGetBlockAttribute(block, attr)

Get a field value inside a block record.

Syntax

```
%if LibIsEqual(LibGetBlockAttribute(ssBlock,"MaskType"), ...
  "Task Block")
  %assign isTaskBlock = 1
%endif
```

Returns

Returns the value of the attribute (field) or an empty string if it does not exist.

See `LibGetBlockAttribute` in matlabroot/rtw/c/tlc/lib/asynclib.tlc.

LibGetCallerClockTickCounter(sfcnBlock)

For use by asynchronous S-functions with function call outputs. Asynchronous tasks can manage their own time. `LibGetCallerClockTickCounter` is used to access an upstream asynchronous task's time counter. This is preferred when being driven by another asynchronous rate (e.g., an Interrupt block driving a Task block) because the time the interrupt occurred is used, rather than the time the task is allowed to run.

Returns

Returns a string for the counter variable for the upstream asynchronous task.

See `LibGetCallerClockTickCounter` in matlabroot/rtw/c/tlc/lib/asynclib.tlc.

LibGetCallerClockTickCounterHigh(sfcnBlock)

For use by asynchronous S-functions with function-call outputs. Asynchronous tasks can manage their own time. `LibGetCallerClockTickCounterHigh` is used to access the high word of an upstream asynchronous task's time counter. This is preferred when being driven by another asynchronous rate (for example, an Interrupt block driving a Task block) because the time the interrupt occurred is used rather than the time the Task is allowed to run.

Returns

Returns a string for the high word of the counter variable for the upstream asynchronous task.

See `LibGetCallerClockTickCounterHigh` in matlabroot/rtw/c/tlc/lib/asynclib.tlc.

LibGetDataTypeComplexNameFromId(id)

Returns the name of the complex data type corresponding to a data type ID. For example, if `id==tSS_DOUBLE` then `LibGetDataTypeComplexNameFromId` returns `"creal_T"`.

See `LibGetDataTypeComplexNameFromId` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibGetDataTypeEnumFromId(id)

Returns the data type `enum` corresponding to a data type ID. For example, if `id==tSS_DOUBLE`, then `enum` is `"SS_DOUBLE"`. If `id` does not correspond to a built-in data type, `LibGetDataTypeEnumFromId` returns `""`.

See `LibGetDataTypeEnumFromId` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibGetDataTypeIdAliasedThruToFromId(id)

Returns the data type `IdAliasedThruTo` that corresponds to a data type ID. For example, if `yourfloat` is an alias to `myfloat`, and `myfloat` is an alias to `double`, then the `IdAliasedThruTo` for both `yourfloat` and `myfloat` is 0 (because the ID for `double` is 0).

See `LibGetDataTypeIdAliasedThruToFromId` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibGetDataTypeIdAliasedToFromId(id)

Returns the data type `IdAliasedTo` that corresponds to a data type ID. For example, if `yourfloat` is an alias to `myfloat`, and `myfloat` is an alias to `double`, then the `IdAliasedTo` for `yourfloat` is the ID for `myfloat`, and the `IdAliasedTo` for `myfloat` is 0 (because the ID for `double` is 0).

See `LibGetDataTypeIdAliasedToFromId` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibGetDataTypeIdResolvesToFromId(id)

Returns the data type `IdResolvesTo` that corresponds to a data type ID.

See `LibGetDataTypeIdResolvesToFromId` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibGetDataTypeNameFromId(id)

Returns the data type name that corresponds to a data type `ID`.

See `LibGetDataTypeNameFromId` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibGetDataTypeSLSizeFromId(id)

Return the size (as Simulink knows it) corresponding to a data type ID.

See `LibGetDataTypeSLSizeFromId` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibGetDataTypeStorageIdFromId(id)

Returns the data type `StorageId` corresponding to a data type ID. For example, if the input ID is the ID for a 16-bit signed fixed-point data type, then the `StorageId` corresponds to `int16`.

See `LibGetDataTypeStorageIdFromId` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibGetFcnCallBlock(sfcnblock,callIdx)

Given an S-function block and call index, return the block record for the downstream function-call subsystem block.

Syntax

```
%assign ssBlock = LibGetFcnCallBlock(block,0)
```

Returns

The block record of the downstream function-call subsystem connected to that element (call index).

See `LibGetFcnCallBlock` in matlabrootrtw/c/tlc/lib/asynclib.tlc.

LibGetRecordDataTypeId(rec)

Returns the data type identifier of the specified record as a an integer.

See `LibGetRecordDataTypeId` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibGetRecordDimensions(rec)

Returns the dimensions of the specified record as a vector of integers.

See `LibGetRecordDimensions` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibGetRecordIsComplex(rec)

Returns the value 1 if the specified record is complex, and zero otherwise.

See `LibGetRecordIsComplex` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibGetRecordWidth(rec)

Returns the width of the specified record as an integer.

See `LibGetRecordWidth` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibGetT()

Returns a string to access the absolute time, which is a floating-point number. However, if you have configured the model for integer-only code generation (by deselecting **Support floating-point numbers**), the string represents an integer equal to the number of base rate ticks (the absolute time divided by the base sample time) rather than the absolute time.

You should use `LibGetT` to access time only.

`LibGetT` is the TLC version of the SimStruct macro `ssGetT`.

See `LibGetT` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibIsComplex(arg)

Returns `1` if the argument passed in is complex, `0` otherwise.

See `LibIsComplex` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibIsFirstInitCond()

`LibIsFirstInitCond` returns generated code intended for placement in the initialization function. This code determines, during run-time, whether the initialization function is being called for the first time.

`LibIsFirstInitCond` also sets a flag that tells the Simulink Coder code generator if it needs to declare and maintain the `first-initialize-condition` flag.

`LibIsFirstInitCond` is the TLC version of the SimStruct macro `ssIsFirstInitCond`.

See `LibIsFirstInitCond` in matlabroot/rtw/c/tlc/lib/syslib.tlc.

LibIsMajorTimeStep()

Returns a string to access whether the current simulation step is a major time step.

`LibIsMajorTimeStep` is the TLC version of the SimStruct macro `ssIsMajorTimeStep`.

See `LibIsMajorTimeStep` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibIsMinorTimeStep()

Returns a string to access whether the current simulation step is a minor time step.

`LibIsMinorTimeStep` is the TLC version of the SimStruct macro `ssIsMinorTimeStep`.

See `LibIsMinorTimeStep` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibManageAsyncCounter(sfcnBlock, callIdx)

For use by asynchronous S-Functions with function-call outputs. Asynchronous tasks can manage their own time, and use `LibManageAsyncCounter` to determine whether a need exists for an asynchronous counter to manage its own timer.

Example

```
%if LibManageAsyncCounter(block,callIdx)
  %%   %<LibSetAsyncCounter(block,callIdx), CodeGetCounter)>
```

where `CodeGetCounter` is target specific code reading hardware timer.

Returns

Returns `TLC_TRUE` if an asynchronous counter is required to manage its own counter, otherwise `TLC_FALSE`.

See `LibManageAsyncCounter` in matlabroot/rtw/c/tlc/lib/asynclib.tlc.

LibMaxIntValue(dtype)

For a built-in integer data type, `LibMaxIntValue` returns the formatted maximum value of that data type.

See `LibMaxIntValue` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibMinIntValue(dtype)

For a built-in integer data type, `LibMinIntValue` returns the formatted minimum value of that data type.

See `LibMinIntValue` in matlabroot/rtw/c/tlc/lib/dtypelib.tlc.

LibNeedAsyncCounter(sfcnBlock, callIdx)

For use by asynchronous S-functions with function-call outputs. Asynchronous tasks use `LibNeedAsyncCounter` to determine whether a need exists for an asynchronous counter.

Example

```
%if LibNeedAsyncCounter(block,0)
  %<LibSetAsyncCounter(block,0), "tickGet()")>
```

Returns

Returns `TLC_TRUE` if an asynchronous counter is required, otherwise `TLC_FALSE`.

See `LibNeedAsyncCounter` in matlabroot/rtw/c/tlc/lib/asynclib.tlc.

LibSetAsyncClockTicks(sfcnBlock, callIdx, buf1, buf2)

For use by asynchronous S-functions with function-call outputs. Asynchronous tasks can manage their own time. Use `LibSetAsyncClockTicks` to return code that sets `clockTick` counters that are to be maintained by the asynchronous task. If the data type of a `clockTick` counter maintained by the asynchronous task is `tSS_TIMER_UINT32_PAIR`, the low and high word of the `clockTick` counter are set.

Arguments

`buf1` — Code that reads the low word of the hardware counter

`buf2` — Code that reads the high word of the hardware counter. Leave `buf2` empty if hardware counter length is less than 32 bits.

Returns

Returns code that sets `clockTick` counters that are to be maintained by the asynchronous task.

Example

```
%if LibNeedAsyncCounter(block, callIdx)
```

```
%<LibSetAsyncCounter(block, 0, buf1, buf2)>
%endif
```

See `LibSetAsyncClockTicks` in matlabroot/rtw/c/tlc/lib/asynclib.tlc.

LibSetAsyncCounter(sfcnBlock, callIdx, buf)

For use by asynchronous S-functions with function-call outputs. Asynchronous tasks can manage their own time. Use `LibSetAsyncCounter` to return code that sets a counter variable that is to be maintained by the asynchronous task.

Returns

Returns code that sets the counter variable that is to be maintained by the asynchronous task.

Example

```
%if LibNeedAsyncCounter(block,0)
  %<LibSetAsyncCounter(block,0, "tickGet()")>
%endif
```

See `LibSetAsyncCounter` in matlabroot/rtw/c/tlc/lib/asynclib.tlc.

LibSetAsyncCounterHigh(sfcnBlock, callIdx, buf)

For use by asynchronous S-functions with function-call outputs. Asynchronous tasks can manage their own time. Use `LibSetAsyncCounterHigh` to return code that sets the higher word of the counter variable that is to be maintained by the asynchronous task.

Returns

Returns code that sets the higher word of the counter variable that is to be maintained by the asynchronous task.

Example

```
%if LibNeedAsyncCounter(block,0)
%<LibSetAsyncCounterHigh(block,0, "hightTickGet()")>
%endif
```

See `LibSetAsyncCounterHigh` in matlabroot/rtw/c/tlc/lib/asynclib.tlc.

LibTIDInSystem(system, fcnType)

Purpose

Returns a task identifier (TID) if it is in the scope of a subsystem function and can be called before or during TLC generating code.

Syntax

```
LibTIDInSystem(system, fcnType)
```

Arguments

system

A record within the global CompiledModel record.

fcnType

One of the following: `'Output'`, `'Update'`, `'Outputupdate'`.

Description

This function returns a TID if it is in the scope of a subsystem function and can be called before or during TLC generating code. It returns the TID argument name, if a TID is passed as argument in the system function scope. If TID is not passed as argument in the scope, this function returns:

- '0' if model is single tasking.
- The TID value of the subsystem if the subsystem is single rate.
- A local TID variable name, if the subsystem is multirate. A local TID variable is added to the subsystem code.

Note: This function issue an error message if it is called for a reusable subsystem whose instances run at different rates.

See `LibTIDInSystem` in matlabroot/rtw/c/tlc/lib/syslib.tlc.

Advanced Functions

In this section...

LibAppendToModelReferenceUserData(data)

Appends the given data object to the user data in the `binfo` file for the model currently being built. This function is only called during Simulink Coder builds for model reference targets.

Note: The data argument cannot be a vector or matrix. To work around this limitation, create a record with a field containing the vector or matrix data and pass this record into the function.

See `LibAppendToModelReferenceUserData` in matlabroot/rtw/c/tlc/mw/modelrefutil.tlc.

LibBlockInputSignalBufferDstPort(portIdx)

Returns the output port corresponding to input port (`portIdx`) that share the same memory, otherwise (-1) is returned. You will need to use `LibBlockInputSignalBufferDstPort` when you specify `ssSetInputPortOverWritable(S,portIdx,TRUE)` in your S-function.

If an input port and some output port of a block are

- Not test points, and
- The input port is overwritable

then the output port might reuse the same buffer as the input port. In this case, `LibBlockInputSignalBufferDstPort` returns the index of the output port that reuses the specified input port's buffer. If none of the block's output ports reuse the specified input port buffer, then `LibBlockInputSignalBufferDstPort` returns `-1`.

`LibBlockInputSignalBufferDstPort` is the TLC version of the Simulink macro `ssGetInputPortBufferDstPort`.

Example

Assume you have a block that has two input ports, both of which receive a complex number in 2-wide vectors. The block outputs the product of the two complex numbers.

```
%assign u1r = LibBlockInputSignal (0, "", "", 0)
%assign u1i = LibBlockInputSignal (0, "", "", 1)
%assign u2r = LibBlockInputSignal (1, "", "", 0)
%assign u2i = LibBlockInputSignal (1, "", "", 1)
%assign yr  = LibBlockOutputSignal (0, "", "", 0)
%assign yi  = LibBlockOutputSignal (0, "", "", 1)
%if (LibBlockInputSignalBufferDstPort(0) != -1)
  %% The first input is going to be overwritten by yr so
  %% we need to save the real part in a temporary variable.
  {
  real_T tmpRe = %<u1r>;
%assign u1r = "tmpRe";
%endif
%<yr> = %<u1r> * %<u2r> - %<u1i> * %<u2i>;
```

```
%<yi> = %<u1r> * %<u2i> + %<u1i> * %<u2r>;
%if (LibBlockInputSignalBufferDstPort(0) != -1)
  }
%endif
```

Note that, because only one output port exists, this example could have used `(LibBlockInputSignalBufferDstPort(0) == 0)` as the Boolean condition for the `%if` statements.

See `LibBlockInputSignalBufferDstPort` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalStorageClass(portIdx, sigIdx)

Returns the storage class of the specified block input port signal. The storage class can be `"Auto"`, `"ExportedSignal"`, `"ImportedExtern"`, or `"ImportedExternPointer"`.

See `LibBlockInputSignalStorageClass` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockInputSignalStorageTypeQualifier (portIdx, sigIdx)

Returns the storage type qualifier of the specified block input port signal. The type qualifier can be anything entered by the user, such as `const`. The default type qualifier is `"Auto"`, which means do the default action.

See `LibBlockInputSignalStorageTypeQualifier` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalIsGlobal(portIdx)

Returns 1 if the specified block output port signal is declared in the global scope, otherwise returns `0`.

If `LibBlockOutputSignalIsGlobal` returns `1`, then the variable holding this signal is accessible from anywhere in generated code. For example, `LibBlockOutputSignalIsGlobal` returns `1` for signals that are test points, external, or invariant.

See `LibBlockOutputSignalIsGlobal` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalIsInBlockIO(portIdx)

Returns 1 if the specified block output port exists in the global block I/O data structure. You might need to use this if you specify `ssSetOutputPortReusable(S,portIdx,TRUE)` in your S-function.

See matlabroot/toolbox/simulink/simdemos/simfeatures/tlc_c/sfun_multiport.tlc.

See `LibBlockOutputSignalIsInBlockIO` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalIsValidLValue(portIdx)

Returns `1` if the specified block output port signal can be used as a valid left-side argument (`lvalue`) in an assignment expression, otherwise returns `0`. For example, `LibBlockOutputSignalIsValidLValue` returns `1` if the block output port signal is in read/write memory.

See `LibBlockOutputSignalIsValidLValue` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalStorageClass(portIdx)

Returns the storage class of the block's specified output signal. The storage class can be `"Auto"`, `"ExportedSignal"`, `"ImportedExtern"`, or `"ImportedExternPointer"`.

See `LibBlockOutputSignalStorageClass` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockOutputSignalStorageTypeQualifier (portIdx)

Returns the storage type qualifier of the block's specified output signal. The type qualifier can be anything entered by the user, such as `const`. The default type qualifier is `Auto`, which means do the default action.

See `LibBlockOutputSignalStorageType` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockSrcSignalBlock(portIdx, sigIdx)

Returns a reference to the block that is the source of the specified block input port element. The return argument is one of the following:

`[systemIdx, blockIdx]`	If the block output or state is unique
`"ExternalInput"`	If external input (root inport)
`"Ground"`	If unconnected or connected to ground
`"FcnCall"`	If function-call output
`0`	If not unique (i.e., source for a Merge block or a signal reused because of block I/O optimization)

Example

The following code fragment finds the block that drives the second input on the first port of the current block, then assigns the input signal of this source block to the variable `y`:

```
%assign srcBlock = LibBlockSrcSignalBlock(0, 1)
%% Make sure that the source is a block
%if TYPE(srcBlock) == "Vector"
  %assign sys = srcBlock[0]
  %assign blk = srcBlock[1]
  %assign block = CompiledModel.System[sys].Block[blk]
  %with block
    %assign u = LibBlockInputSignal(0, "", "", 0)
    y = %<u>;
  %endwith
%endif
```

See `LibBlockSrcSignalBlock` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockSrcSignalIsDiscrete(portIdx, sigIdx)

Returns 1 if the source signal corresponding to the specified block input port element is discrete, otherwise returns 0.

Note that `LibBlockSrcSignalIsDiscrete` also returns 0 if the driving block cannot be uniquely determined to be a merged or reused signal (i.e., the source is a Merge block or the signal has been reused because of optimization).

See `LibBlockSrcSignalIsDiscrete` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockSrcSignalIsGlobalAndModifiable (portIdx, sigIdx)

`LibBlockSrcSignalIsGlobalAndModifiable` returns 1 if the source signal corresponding to the specified block input port element satisfies the following three conditions:

- It is readable everywhere in the generated code.
- It can be referenced by its address.
- Its value can change (i.e., it is not declared as a `const`).

Otherwise, `LibBlockSrcSignalIsGlobalAndModifiable` returns `0`.

See `LibBlockSrcSignalIsGlobalAndModifiable` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibBlockSrcSignalIsInvariant(portIdx, sigIdx)

Returns `1` if the source signal corresponding to the specified block input port element is invariant (i.e., the signal does not change).

For example, a source block with a constant `TID` (or equivalently, an infinite sample time) would output an invariant signal.

See `LibBlockSrcSignalIsInvariant` in matlabroot/rtw/c/tlc/lib/blkiolib.tlc.

LibGetModelReferenceUserData(modelName)

Gets the user data for the given model. This returns a vector with one element for each time `LibAppendToUserData` is called in the given model. This function cannot be called during builds where the target type is SIM.

See `LibGetModelReferenceUserData` in matlabroot/rtw/c/tlc/mw/modelrefutil.tlc.

LibGetReferencedModelNames()

Gets the names of the models referenced by the model that is currently being built. Returns the data as a structure with two fields:

- `NumReferencedModels`: an integer with the number of model names
- `ReferencedModel`: an array of structures, where each structure has a field –`Name`– containing the name of a referenced model

See `LibGetReferencedModelNames` in matlabroot/rtw/c/tlc/mw/modelrefutil.tlc.

LibIsModelReferenceRTWTarget()

Returns true if theSimulink Coder build process is generating code for model reference Simulink Coder target.

See `LibIsModelReferenceRTWTarget` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibIsModelReferenceSimTarget()

Return true if we are generating code for model reference Simulation target.

See `LibIsModelReferenceSimTarget` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

LibIsModelReferenceTarget()

Return true if we are generating code for model reference target.

See `LibIsModelReferenceTarget` in matlabroot/rtw/c/tlc/lib/utillib.tlc.

A

TLC Error Handling

Generating Errors from TLC Files

In this section...

TLC Error Generation Overview

To generate errors from TLC files, you can use the `%exit` directive. Alternatively, you can use one of the library functions described below that calls `%exit` for you. The two types of errors are

Usage errors	These can be caused by incorrect models.
Fatal (internal) TLC coding errors	These *cannot* be caused by incorrect models.

Usage Errors

Usage errors are errors resulting from incorrect models or attributes defined on a model. For example, suppose you have an S-Function block and an inline TLC file for a specific D/A device. If a model can contain only one copy of this S-function, then an error needs to be generated for a model that contains two copies of this S-Function block.

Using Library Functions

To generate usage errors related to a specific block, use the library function

```
LibBlockReportError(block,"error string")
```

The `block` argument is the block record if it isn't scoped. If the block is currently scoped, then you can specify `block` as [ ].

To generate general usage errors that are not related to a specific block, use

```
LibReportError("error string")
```

These library functions prefix the string `Simulink Coder Error:` to the message you provide when reporting the error.

For a usage example of these functions, refer to `gensfun.tlc` for block errors and `commonsetup.tlc` for common errors. There are other files that use these functions in the TLC source folders within *`matlabroot`*`/rtw/c/tlc`.

Fatal (Internal) TLC Coding Errors

Suppose you have an S-function that has a local function that can accept only numerical numbers. You might want to add an *assert* requiring that the inputs be only numerical numbers. These asserts can indicate fatal coding errors for which the user does not have a way of building a model or specifying attributes that can cause the error to occur.

Using Library Functions

The two available library functions are

```
LibBlockReportFatalError(block,"fatal coding error message")
```

where *`block`* is the offending block record (or `[]` if the block is already scoped), and

```
LibReportFatalError("fatal coding error message")
```

for error messages that are not block specific. For example, to add assert code you could use

```
%if TYPE(argument) != "Number"
  %<LibBlockReportFatalError(block,"unexpected argument type")>
%endif
```

These library functions prefix the string `Simulink Coder Fatal:` to the message you provide and display the call stack when reporting the error.

For a usage example of these functions, refer to `gensfun.tlc` for block errors and `commonsetup.tlc` for common errors. There are other files that use these functions in the folder *`matlabroot`*`/rtw/c/tlc`.

Using %exit

You can call `%exit` to generate fatal error messages. However, MathWorks suggests that you use one of the library functions described above.

When generating fatal error messages directly with `%exit`, it is good practice to give a stack trace with the error message. This lets you see the call chain of functions that caused the error. To generate a stack trace, generate the message using the format:

```
%setcommandswitch "-v1"
%exit Simulink Coder Fatal: error string
```

Formatting Error Messages

If you want to display a formatted, multiple-line error message, create a local variable that contains the message text. For example:

```
%openfile message
My message text
with newlines
%closefile message
```

After formatting your error message, use one of the error reporting library functions described above, such as `LibReportError`, to report your error when it occurs. For example:

```
%<LibReportError(message)>
```

The error reporting library functions provide an error message prefix, such as `Simulink Coder Error:`.

Testing Error Messages

It is strongly suggested that you test your error messages before releasing your new TLC code. To test your error messages, copy the relevant code into a `test.tlc` file and run

```
tlc test.tlc
```

at the MATLAB prompt.

TLC Error Messages

In this section...

Using TLC Error Messages to Troubleshoot

This section lists and describes error messages generated by the Target Language Compiler. Use this reference to

- Confirm that an error has been reported
- Determine possible causes for an error
- Determine possible ways to fix an error

Alphabetical List of Error Messages

%closefile or %selectfile or %flushfile argument must be a valid open file

In `%closefile` or `%selectfile` or `%flushfile`, the argument must be a valid file variable opened with `%openfile`.

%define no longer supported, use %function instead

Macros are not supported. You must rewrite macros as functions or inline them in your code.

%error directive: *`text`*

Code containing the `%error` directive generates this message. It normally indicates some condition that the code was unable to handle and displays the text following the `%error` directive.

%exit directive: *`text`*

Code containing the `%exit` directive causes this message. It typically indicates some condition that the code was unable to handle and displays the text following the `%exit`

directive. Note that this directive causes the Target Language Compiler to terminate regardless of the `-mnumber` command-line option.

%filescope has already been used in this file

The user attempted to use the `%filescope` directive more than once in a file.

%trace directive: *text*

The `%trace` directive produces this error message and displays the text following the `%trace` directive. Trace directives are reported only when the `-v` option (verbose mode) appears on the command line. Note that `%trace` directives are not considered errors and do not cause the Target Language Compiler to stop processing.

%warning directive: *text*

The `%warning` directive produces this error message and displays the text following the `%warning` directive. Note that `%warning` directives are not considered errors and do not cause the Target Language Compiler to stop processing.

A %implements directive must appear within a block template file and must match the %language and type specified

A block template file was found, but it did not contain an `%implements` directive. An `%implements` directive is required so that the expected language and type are implemented by this block template file. See "Object-Oriented Facility for Generating Target Code" on page 6-32 for more information.

A %switch statement can only have one %default

The user has written a `%switch` statement with multiple `%default` cases, as in the following example:

```
%switch expr
  %case 1
    code...
    %break
  %default

more code...
    %break
  %default    %% error
    even more code...
    %break
```

```
%endswitch
```

A language choice must be made using the %language directive prior to using GENERATE or GENERATE_TYPE

To use the `GENERATE` or `GENERATE_TYPE` built-in functions, the Target Language Compiler requires that you first specify the language being generated. This causes the block-level target file to implement the same language and type as specified in the `%language` directive.

A non-homogeneous vector was passed to GENERATE_FORMATTED_VALUE

The built-in `GENERATE_FORMATTED_VALUE` can process only vectors that have homogeneous elements (that is, vectors in which all the elements have the same type).

Ambiguous reference to *`identifier`* — must use array index to refer to one of multiple scopes

In a repeated scope identifier from a database file, you must specify an index to disambiguate the reference. For example

```
Database file:
block
{
 Name    "Abc2"
 Parameter {
  Name   "foo"
  Value   2
 }
}
block
{
 Name    "Abc3"
 Parameter {
  Name   "foo"
  Value   3
 }
}
TLC file:
%<GETFIELD(block, "Name")>
```

In the preceding example, the reference to `block` is ambiguous because multiple repeated scopes named `block` appear in the database file. Use an index to disambiguate the references, as in:

```
%<GETFIELD(block[0], "Name")>
```

An %if statement can only have one %else

The user has written an `%if` statement with multiple `%else` blocks, as in the following example:

```
%if expr
  code...
%else
  more code...
%else          %% error
  even mode code...
%endif
```

Argument to *`identifier`* must be a string

The following built-in functions expect a string and report this error if the argument passed is not a string.

CAST	GENERATE_FILENAME
EXISTS	GENERATE_FUNCTION_EXISTS
FEVAL	GENERATE_TYPE
FILE_EXISTS	GET_COMMAND _SWITCH
FORMAT	IDNUM
GENERATE	SYSNAME

Arguments to *`directive`* must be records

Arguments to `%mergerecord` and `%copyrecord` must be records. Also, the first argument to the following built-in functions must be a record:

- ISALIAS
- REMOVEFIELD
- FIELDNAMES
- ISFIELD
- GETFIELD
- SETFIELD

Arguments to TLC from the MATLAB command line must be strings

An attempt was made to invoke the Target Language Compiler from MATLAB, but some of the arguments that were passed were not strings.

Assertion failed

An expression in an `%assert` statement evaluated to false.

Assignment to scope *`identifier`* is only allowed when using the + operator to add members

Scope assignment must be `scope = scope + variable`.

Attempt to define a function *identifier* on top of an existing variable or function

A function cannot be defined twice. Make sure that you don't have the same function defined in separate TLC files.

Attempt to divide by zero

The Target Language Compiler does not allow division by zero.

Bad cast - unable to cast this expression to *`type`*

The Target Language Compiler cannot cast this expression from its current type to the specified type. For example, the Target Language Compiler cannot cast a string to a number, as in

```
%assign x = "1234"
%assign y = CAST("Number", x );
```

Bad directory (*`dirname`*) in O: *`filename`*

The `-O` option did not specify a valid folder.

`builtin` was expecting expression of type *`type`*, got one of type *`type`*

A built-in was passed an expression of incorrect type.

Cannot %undef any builtin functions or variables

User is not allowed to undefine a TLC built-in or variable. For example

```
%undef FORMAT  %% error
```

Cannot convert string *your_string* to a number

Cannot convert the string to a number.

Changing value of *identifier* from the RTW file

You have overwritten the value that appeared in the `.rtw` file.

Error opening *filename*

The Target Language Compiler could not open the file specified on the command line.

Error writing to file *error*

There was an error while writing to the current output stream; `error` contains the system specific error message.

Errors occurred — aborting

This error message is the last error to be reported. It occurs when either

- The number of error messages exceeds the error message threshold (5 by default).
- Processing completes and errors have occurred.

Expansion directives %<> cannot be nested

It is illegal to nest expansion directives. For example,

```
%<foo(%<expr>)>
```

Instead, do the following:

```
%assign tmp = %<expr>
%<foo(tmp)>
```

Expansion directives %<> cannot span multiple lines; use \ at end of line

An expansion directive cannot span multiple lines. To work around this restriction, use the \ line continuation character. The following is incorrect:

```
%<CompiledModel.System[Sysidx].Block[BlkIdx].Name +
"Hello">
```

Instead, use:

```
%<CompiledModel.System[Sysidx].Block[BlkIdx].Name + \
"Hello">
```

Extra arguments to the *function-name* built-in function were ignored (Warning)

The following built-in functions report this warning when too many arguments are passed to them:

CAST	NUMTLCFILES
EXISTS	OUTPUT_LINES
FILE_EXISTS	SIZE
FORMAT	STRING
GENERATE_FILENAME	STRINGOF
GENERATE_FUNCTION_EXISTS	SYSNAME
IDNUM	TLCFILES
ISFINITE	TYPE
ISINF	WHITE_SPACE
ISNAN	WILL_ROLL

File name too long (directory =*dirname*, name =*filename*)

The specified *filename* was too long. The default limits are 256 characters for *filename* and 1024 characters for *dirname*, but the limits can be larger, depending on the platform.

format is not a legal format value

The specified format was not legal for the `%realformat` directive. Valid format strings are `"EXPONENTIAL"` and `"CONCISE"`.

Function argument mismatch; function *function_name* expects *number* arguments

When calling a function, too many or too few arguments were passed to it.

Function reached the end and did not return a value

Functions that are not declared as `void` or `Output` must return a value. If a return value is not desired, declare the function as `void`, otherwise make it return a value.

Function values are not allowed

Attempt to use a TLC function as a variable.

Identifier *`identifier`* multiply defined. Second and succeeding definitions ignored.

The user is attempting to add the same field to a record more than once, as in the following code.

```
%createrecord err { foo 1; rec { val 2 } }
%addtorecord err foo 2             %% error
```

Identifier *`identifier`* used on a %foreach statement was already in scope (Warning)

The argument to a `%foreach` statement cannot be defined prior to entering the `%foreach`.

Illegal use of eval (i.e., %<...>)

It is illegal to use evals in `.rtw` files. There are also some places where evals are not allowed in directives. For example:

```
%function %<foo>(a, b, c) void  %% error
%endfunction
```

Indices may not be negative

An index used in a [ ] expression must be a nonnegative integer.

Indices must be constant integral numbers

An index used in a [ ] expression must be an integer number.

Invalid handle

An invalid handle was passed to the Target Language Compiler server mode.

Invalid identifier range, the leading strings *`string1`* and *`string2`* must match

In a range of signals, for example, `u1:u10`, the identifier in the first argument did not match the identifier in the second.

Invalid identifier range, the lower bound (*`bound`*) must be less than the upper bound (*`bound`*)

In a range of signals, for example, `u1:u10`, the lower bound was higher than the upper bound.

Invalid type for unary *operator*

Unary operators – and + require numeric types. Unary operator – requires an integral type. Unary operator `!` requires a numeric type.

Invalid type *type*

An invalid type was passed to a built-in function.

It is illegal to return a function from a function

A function value cannot be returned from a function call.

Named value *identifier* already exists within this *scope-identifier*; use %assign to change the value

You cannot use the block addition operator + to add a value that is already a member of the indicated block. Use `%assign` to change the value of an existing value. This example produces this error:

```
%assign x = BLK { a 1; b 2 }
%assign a = 3
%assign x = x + a
```

Use this instead:

```
%assign x.a = 3
```

No %case statement(s) seen yet, statement ignored

Statements that appear inside a `%switch` statement but precede `%case` statements are ignored, as in the following code:

```
%switch expr
%assign x = 2  %% this statement will be ignored
  %case 1
    code
    %break
%endswitch
```

Only double and character arrays can be converted from MATLAB to TLC. This can occur if the MATLAB function does not return a value (see %matlab)

Only double and character arrays can be converted from MATLAB to the Target Language Compiler. This error can occur if the MATLAB function does not return a value (see `%matlab`). For example,

```
%assign a = FEVAL("int8",3)
%matlab disp(a)
```

Only one output is allowed from the TLC

An attempt was made to receive multiple outputs from the MATLAB version of the Target Language Compiler.

Only strings of length 1 can be assigned using the [ ] notation

The right-hand side of a string assignment using the [ ] operator must be a string of length 1. You can replace only a single character using this notation.

Only strings or cells of strings may be used as the argument to Query and ExecString

A cell containing nonstring data was passed as the third argument to `Query` or `ExecString` in server mode.

Only vectors of the same length as the existing vector value can be assigned using the [ ] notation

In the [ ] notation to replace a row of a matrix, the row must be a vector of the same length as the existing rows.

Output file *identifier* opened with %openfile was not closed

Output files opened with `%openfile` must be closed with `%closefile`. The *identifier* is the name of the variable specified in the `%openfile` directive.

Note This might also occur a syntax error is present in a code section between an `openfile` and `closefile`, or if you try to assign the output of a function of type `void` or `Output` to a variable.

Ranges, identifier ranges, and repeat values cannot be repeated

You cannot repeat a range, identifier range, or repeat value. This prevents things like `[1@2@3]`.

String cannot modify the setting for the command line switch '-*switch*'

`%setcommandswitch` does not recognize the specified switch, or cannot modify it (e.g., `-r` cannot be modified).

string is not a recognized user defined property of this handle

The query performed on a TLC server mode handle is looking for an undefined property.

Syntax error

The indicated line contains a syntax error, for more information on syntax, see "Directives and Built-In Functions".

The %break directive can only appear within a %foreach, %for, %roll, or %switch statement

The `%break` directive can be used only in a `%foreach`, `%for`, `%roll`, or `%switch` statement.

The %case and %default directives can only be used within the %switch statement

A `%case` or `%default` directive can appear only within a `%switch` statement.

The %continue directive can only appear within a %foreach, %for, or %roll statement

The `%continue` directive can be used only in a `%foreach`, `%for`, or `%roll` statement.

The %foreach statement expects a constant numeric argument

The argument of a `%foreach` must be a numeric type. For example:

```
%foreach Index = [1 2 3 4]
...
%endforeach
```

`%foreach` cannot accept a vector as input.

The %if statement expects a constant numeric argument

The argument of an `%if` statement must be a numeric type. For example,

```
%if [ 1 2 3 ]
...
%endif
```

`%if` cannot accept a vector as input.

The %implements directive expects a string or string vector as the list of languages

You can use the `%implements` directive to specify a string for the language being implemented, or to indicate that it implements multiple languages by using a vector of strings. You cannot specify any other argument type to the `%implements` directive.

The %implements directive specifies type as the *type* where *type* was expected

The type specified in the `%implements` directive must exactly match the type specified in the block or on the `GENERATE_TYPE` directive. If you want to specify that the block accept multiple input types, use the `%implements *` directive, as in

```
%implements * "C"   %% I accept any type and generate C code
```

The %implements language does not match the language currently being generated (*language*)

The language or languages specified in the `%implements` directive must exactly match the `%language` directive.

The %return statement can only appear within the body of a function

A `%return` statement can be only in the body of a function.

The == and != operators can only be used to compare values of the same type

The == and != operator arguments must be the same type. You can use the `CAST()` built-in function to change them into the same type.

The argument for %openfile must be a valid string

When you open an output file, the name specified for the file must be a valid string.

The argument for %with must be a valid scope

The argument to `%with` must be a valid scope identifier. For example,

```
%assign x = 1
%with x
...
%endwith
```

In this code, the `%with` statement argument is a number and produces this error message.

The argument for an [ ] operation must be a repeated scope symbol, a vector, or a matrix

When you use the [ ] operator to index, the expression on the left of the brackets must be a vector, matrix, string, numeric constant, or a repeated scope identifier. When you use array indexing on a scalar, the constant is automatically scalar expanded and the value of the scalar is returned. For example,

```
%openfile x
%assign y = x[0]
```

This example causes this error because `x` is a file and is not valid for indexing.

The argument to %addincludepath must be a valid string

The argument to `%addincludepath` must be a string.

The argument to %include must be a valid string

The argument to the input file control directive must be a valid string with the filename given in double quotation marks.

The begin directive must be in the same file as the corresponding end directive.

These Target Language Compiler `begin` directives must appear in the same file as their corresponding `end` directives: `%function`, `%switch`, `%foreach`, `%roll`, and `%for`. Place the construct entirely within one Target Language Compiler source file.

The *`begin`* directive on this line has no matching *`end`* directive

For block-scoped directives, this error is produced if a matching `end` directive is missing. This error can occur for the following block-scoped Target Language Compiler directives.

Begin Directive	End Directive	Description
`%if`	`%endif`	Conditional inclusion
`%for`	`%endfor`	Looping
`%foreach`	`%endforeach`	Looping
`%roll`	`%endroll`	Loop rolling
`%with`	`%endwith`	Scoping directive
`%switch`	`%endswitch`	Switch directive
`%function`	`%endfunction`	Function declaration directive
`{`	`}`	Record creation

The error is reported on the line that opens the scope and does not have matching `end` scope.

Note Nested scopes must be closed before their parent scopes. Failure to include an `end` for a nested scope often causes this error, as in

```
%if Block.Name == "Sin 3"
    %foreach idx = Block.Width %endif
%% Error reported here that the %foreach was not terminated
```

The construct %matlab *`function_name`*(...) construct is illegal in standalone tlc

You cannot call MATLAB from stand-alone TLC.

The FEVAL() function can accept only 2-dimensional arrays from MATLAB, not number dimensions

Return values from MATLAB can have at most two dimensions.

The FEVAL() function can accept vectors of numbers or strings only when calling MATLAB

Vectors passed to MATLAB can be numbers or strings. See "FEVAL Function" on page 6-44.

The FEVAL() function requires the name of a function to call

FEVAL requires a function to call. This error appears only inside MATLAB.

The final argument to %roll must be a valid block scope

When you use `%roll`, the final argument (prior to extra user-specified arguments) must be a valid block scope. See "%roll" on page 6-29 for a description of this command.

The first argument of a ? : operator must be a Boolean expression

The `? :` operator must have a Boolean expression as its first operand.

The first argument to GENERATE or GENERATE_TYPE must be a valid scope

When you call GENERATE or GENERATE_TYPE, the first argument must be a valid scope. See the "GENERATE and GENERATE_TYPE Functions" on page 6-33 for more information and examples.

The function *name* requires at least *number* arguments

User is passing too few arguments to a function, as in the following code:

```
%function foo(a, b, c)
  %return a + b + c
%endfunction

%<foo(1, 2)>  %% error
```

The GENERATE function requires at least 2 arguments

When you call the GENERATE built-in function, the first two arguments must be the block and the name of the function to call.

The GENERATE_TYPE function requires at least 3 arguments

When you call the `GENERATE_TYPE` built-in function, the first three arguments must be the block, the name of the function to call, and the type.

The ISINF(), ISNAN(), ISFINITE(), REAL(), and IMAG() functions expect a real or complex valued argument

These functions expect a `Real` or complex value as the input argument.

The language being implemented cannot be changed within a block template file

You cannot change the language using the `%language` directive within a block template file.

The language being implemented has changed from *`old-language`* to *`new-language`* (Warning)

The language being implemented should not be changed in midstream because `GENERATE` function calls that appear prior to the `%language` directive can cause `generate` functions to load for the prior language. Only one language directive should appear in a given file.

The left-hand side of a . operator must be a valid scope identifier

When you use the `.` operator, the left-hand side of the `.` operator must be a valid in-scope identifier. For example:

```
%assign x = 1
%assign y = x.y
```

In this code, the reference to `x.y` produces this error message because `x` is not defined as a scope.

The left-hand side of an assignment must be a simple expression comprised of ., [ ], and identifiers

Illegal left-hand side of assignment.

The number of columns specified (*`specified-columns`*) did not match the actual number of columns in the rows (*`actual-columns`*)

When you specify a Target Language Compiler matrix, the number of columns specified must match the actual number of columns in the matrix. For example,

```
%assign mat = Matrix(2,1) [[1,2];[2,3]]
```

In this case, the number of columns in the declaration of the matrix (1) did not match the number of columns seen in the matrix (2). Either change the number of columns in the matrix, or change the matrix declaration.

The number of rows specified (*specified-rows*) did not match the actual number of rows seen in the matrix (*actual-rows*)

When you specify a Target Language Compiler matrix, the number of rows must match the actual number of rows in the matrix. For example,

```
%assign mat = Matrix(1,2) [[1,2];[2,3]]
```

In this case, the number of rows in the declaration of the matrix (1) did not match the number of rows seen in the matrix (2). Either change the number of rows in the matrix or change the matrix declaration.

The *operator_name* operator only works on Boolean arguments

The && and || operators work only on Boolean values.

The *operator_name* operator only works on integral arguments

The &, ^, |, <<, >> and % operators work on numbers only.

The *operator_name* operator only works on numeric arguments

The arguments to the following operators both must be either numeric or real: <, <=, >, >=, -, *, /. This error can also occur when you use + as a unary operator. In addition, the FORMAT built-in function expects either a numeric or real argument.

The return value from the RollHeader function must be a string

When you use `%roll`, the `RollHeader()` function specified in `Roller.tlc` must return a string value. See "%roll" on page 6-29 for a complete discussion of the `%roll` construct.

The roll argument to %roll must be a nonempty vector of numbers or ranges

When you use `%roll`, the `roll` vector cannot be empty and must contain numbers or ranges of numbers. See "%roll" on page 6-29 for a complete discussion of the `%roll` construct.

The second value in a Range must be greater than the first value

In a range, for example, `1:10`, the lower bound was higher than the upper bound.

The specified index (*`index`*) was out of the range *`0 - number-of-elements - 1`*

This error occurs when you index into a nonscalar beyond the end of the variable. For example:

```
%assign x = [1 2 3]
%assign y = x[3]
```

This example would cause this error. Remember, in the Target Language Compiler, array indices start at 0 and go to the number of elements minus 1.

The STRINGOF built-in function expects a vector of numbers as its argument

The `STRINGOF` function expects a vector of numbers. The function treats each number as the ASCII value of a valid character.

The SYSNAME built-in function expects an input string of the form *`xxx/yyy`*

The `SYSNAME` function takes a single string of the form `xxx/yyy` as it appears in the `.rtw` file and returns a vector of two strings, `xxx` and `yyy`. If the input argument does not match this format, `SYSNAME` returns this error.

The threshold on a %roll statement must be a single number

When you use `%roll`, the roll threshold specified must be a single number. See "%roll" on page 6-29 for a complete discussion of the `%roll` construct.

The use of *`feature`* is being deprecated and will not be supported in future versions of TLC. See the TLC manual for alternatives.

The `%define` and `%generate` directives are not recommended, as they are being replaced.

The WILL_ROLL built in function expects a range vector and an integer threshold

The `WILL_ROLL` function requires both arguments cited in the message.

There are no more free contexts. Use tlc('close', HANDLE) to free up a context

The global context table has filled up while the TLC server mode is in use.

There was no type associated with the given block for GENERATE

The scope specified to `GENERATE` must include a `Type` parameter that indicates which template file should be used to generate code for the specified scope. For example:

```
%assign scope = block { Name "foo" }
%<GENERATE( scope, "Output" )>
```

This example produces the error message because the scope does not include the parameter `Type`. See the "GENERATE and GENERATE_TYPE Functions" on page 6-33 for more information and examples.

This assignment would overwrite an identifier-value pair from the RTW file. To avoid this error either qualify the left-hand side, or choose another identifier.

The user is trying to modify a field of a record in a `%with` block without qualifying the left-hand side, as in this example:

```
  %createrecord foo { field 1 }
  %with foo
    %assign field = 2  %% error
  %endwith
```

Instead, use

```
%createrecord foo { field 1 }
  %with foo
    %assign foo.field = 2
  %endwith
```

TLC has leaked *`number`* symbols. You may have created a cyclic record. If this not the case then please report this leak to The MathWorks.

There has been a memory leak while running TLC. The most common cause of this is having cyclic records.

Unable to find *identifier* within the *scope-identifier* scope

The given identifier was not found in the scope specified. For example,

```
%assign scope = ascope { x 5 }
%assign y = scope.y
```

In this code, the reference to `scope.y` produces this error message.

Unable to open %include file *filename*

The file included in an `%include` directive was not found on the path. Either move the file to a location on the current path, or use the `-I` command-line option or the `%addincludepath` directive to specify the folder that contains the file.

Unable to open block template file *filename* from GENERATE or GENERATE_TYPE

You specified `GENERATE` but the given filename was not found on the Target Language Compiler path. You can

- Add the file to a folder on the path.
- Use the `%generatefile` directive to specify an alternative filename for this block type that is on the path.
- Add the folder in which this file appears to the search path using the `-I` command-line option or the `%addincludepath` directive.

Unable to open output file *filename*

The specified output file could not be opened. Either an invalid filename was specified or the file was read only.

Undefined identifier *identifier_name*

The identifier specified in this expression was undefined.

Unknown type *type* in CAST expression

When you call the `CAST` built-in function, the type must be a valid Target Language Compiler type listed in the table ???.

Unrecognized command line switch passed to string: *switch*

You queried the current state of a switch, but the switch specified was not recognized.

Unrecognized directive *directive-name* seen

An illegal % directive was encountered. The valid directives are shown below.

```
%addincludepath          %addtorecord
%assert                  %assign
%break=                  %case
%closefile               %continue
%copyrecord              %createrecord
%default                 %define
%else                    %elseif
%endbody                 %endfor
%endforeach              %endfunction
%endif                   %endroll
%endswitch               %endwith
%error                   %exit
%filescope               %for
%foreach                 %function
%generate                %generatefile
%if                      %implements
%include                 %language
%matlab                  %mergerecord
%openfile                %realformat
%return                  %roll
%selectfile              %setcommandswitch
%switch                  %trace
%undef                   %warning
%with
```

Unrecognized type *output-type* for function

The function type modifier was not `Output` or `void`. For functions that do not produce output, the default without a type modifier indicates that the function should not produce output.

Unterminated multiline comment.

A multiline comment (i.e., /% %/) does not have a terminator, as in the following code:

```
/% my comment

%assign x = 2
%assign y = x * 7
```

Unterminated string

A string must be closed prior to the end of an expansion directive or the end of a line.

Usage: tlc [options] file

A command-line problem has occurred. The error message contains a list of the available options.

Use of *feature* incurs a performance hit, please see TLC manual for possible workarounds.

The `%undef` and expansion (i.e., `%<expr>`) features can degrade performance.

Value of *type* type cannot be compared

Values of the specified *type* cannot be compared.

Values of *specified_type* type cannot be expanded

The specified type cannot be used on an expansion directive. Files and scopes cannot be expanded. This error can also occur when you expand a function without any arguments. If you use

```
%<Function>
```

call it with the required arguments.

Values of type Special, Macro Expansion, Function, File, Full Identifier, and Index cannot be converted to MATLAB variables

Values of the types listed in the message cannot converted to MATLAB variables.

When appending to a buffer stream, the variable must be a string

You can specify the `append` option for a buffer stream only if the variable currently exists as a string. Do not use the `append` option if the variable does not exist or is not a string. This example produces this error.

```
%assign x = 1
%openfile x , "a"
%closefile x
```

TLC Function Library Error Messages

The functions in the TLC function library can generate many error messages that are not documented. These messages are sufficiently self-descriptive so that they do not need additional explanation. However, if you encounter an error message that does not provide enough description to resolve your problem, contact our technical support staff.