

Real-Time Workshop[®]

For Use with Simulink[®]

- Modeling
- Simulation
- Implementation

Target Language Compiler
Reference Guide

Version 6



How to Contact The MathWorks:



www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup



support@mathworks.com	Technical support
suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information



508-647-7000	Phone
--------------	-------



508-647-7001	Fax
--------------	-----



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

For contact information about worldwide offices, see the MathWorks Web site.

Real-Time Workshop Target Language Compiler Reference Guide

© COPYRIGHT 1997 - 2004 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and TargetBox is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History:	May 1997	First printing	New for Target Language Compiler 1.0
	September 2000	Online only	Updated for Version 4 (Release 12)
	April 2001	Online only	Updated for Version 4.1 (Release 12.1)
	July 2002	Online only	Updated for Version 5.0 (Release 13)
	June 2004	Online only	Updated for Version 6.0 (Release 14)

Introducing the Target Language Compiler

1

What Is the Target Language Compiler?	1-2
Overview of the TLC Process	1-3
Overview of the Code Generation Process	1-5
Target Language Compiler Capabilities	1-7
Customizing Output	1-7
Inlining S-Functions	1-7
Modifying and Diversifying Code Generation	1-8
Code Generation Process	1-9
How TLC Determines S-Function Inlining Status	1-9
A Look at Inlined and Noninlined S-Function Code	1-10
Advantages of Inlining S-Functions	1-13
Motivations	1-13
Inlining Process	1-14
Search Algorithm for Locating Target Files	1-15
Availability for Inlining and Noninlining	1-15
New Features and Compatibility Issues in Versions 4.0, 4.1, and 5.0	1-16
New Features Added in Version 5.0	1-16
New Features Added in Version 4.1	1-17
New Features Added in Version 4.0	1-18
Compatibility Issues	1-19
Where to Go from Here	1-23
Related Manuals	1-23

2

Code Architecture	2-2
model.rtw and Target Language Compiler Overview	2-4
The Target Language Compiler Process	2-4
Inlining S-Function Concepts	2-6
Noninlined S-Function	2-6
Types of Inlining	2-7
Fully Inlined S-Function Example	2-8
Wrapper Inlined S-Function Example	2-10

Code Generation Architecture

3

Build Process	3-2
A Basic Example	3-2
Invoking Code Generation	3-8
The rtwgen Command	3-8
The tlc Command	3-8
Configuring TLC	3-10
Setting Command Line Arguments	3-10
Configuring for TLC Debugging	3-12
Code Generation Concepts	3-13
Output Streams	3-13
Variable Types	3-14
Records	3-14
Record Aliases	3-16
TLC Files	3-18
Available Target Files	3-18
Summary of Target File Usage	3-22

System Target Files	3-23
Block Target Files	3-25
Block Target File Mapping	3-25
Data Handling with TLC: An Example	3-26
Matrix Parameters in Real-Time Workshop	3-26

Contents of model.rtw

4

Model.rtw File Overview	4-2
Using Scopes in the model.rtw File	4-3
Object Information in the model.rtw File	4-6
Using Library Functions to Access model.rtw Contents ..	4-10
Caution Against Directly Accessing Record Fields	4-10
Exception to Using the Library Functions	4-11

Directives and Built-in Functions

5

Compiler Directives	5-2
Syntax	5-2
Comments	5-17
Line Continuation	5-18
Target Language Values	5-19
Target Language Expressions	5-21
Formatting	5-27
Conditional Inclusion	5-28
Multiple Inclusion	5-29
Object-Oriented Facility for Generating Target Code	5-34
Output File Control	5-36
Input File Control	5-37
Asserts, Errors, Warnings, and Debug Messages	5-38

Built-In Functions and Values	5-39
TLC Reserved Constants	5-51
Identifier Definition	5-51
Variable Scoping	5-56
Target Language Functions	5-66
Command Line Arguments	5-71
Filenames and Search Paths	5-73

Debugging TLC Files

6

About the TLC Debugger	6-2
Tips for Debugging TLC Code	6-2
Using the TLC Debugger	6-3
Invoking the Debugger	6-3
TLC Debugger Command Summary	6-5
TLC Coverage	6-9
Using the TLC Coverage Option	6-9
TLC Profiler	6-13
Using the Profiler	6-13

Inlining S-Functions

7

Introduction	7-2
Writing Block Target Files to Inline S-Functions	7-3
Fully Inlined S-Functions	7-3
Function-Based or Wrappered Code Generation	7-3

Inlining C MEX S-Functions	7-5
S-Function Parameters	7-6
A Complete Example	7-7
Inlining M-File S-Functions	7-18
Inlining Fortran (F-MEX) S-Functions	7-20
TLC Coding Conventions	7-24
Block Target File Methods	7-29
Block Target File Mapping	7-29
Block Functions	7-29
Loop Rolling	7-37
Error Reporting	7-40

TLC Function Library Reference

8

Obsolete Functions	8-2
Target Language Compiler Functions	8-4
Common Function Arguments	8-4
Input Signal Functions	8-9
Output Signal Functions	8-21
Parameter Functions	8-26
Block State and Work Vector Functions	8-31
Block Path and Error Reporting Functions	8-35

Code Configuration Functions	8-37
Sample Time Functions	8-59
Other Useful Functions	8-67
Advanced Functions	8-78

TLC Error Handling

A

Generating Errors from TLC-Files	A-2
Usage Errors	A-2
Fatal (Internal) TLC Coding Errors	A-2
Formatting Error Messages	A-4
TLC Error Messages	A-5
TLC Function Library Error Messages	A-30

Using TLC with Emacs

B

The Emacs Editor	B-2
Getting Started	B-2
Creating a TAGS File	B-2

Index

Introducing the Target Language Compiler

What Is the Target Language Compiler? (p. 1-2)	Overview of the role of the Target Language Compiler in code generation
Target Language Compiler Capabilities (p. 1-7)	Reasons and circumstances for customizing generated code
Code Generation Process (p. 1-9)	Block and system target files, exemplified by inlined S-functions
Advantages of Inlining S-Functions (p. 1-13)	When, how, and why to inline S-functions
New Features and Compatibility Issues in Versions 4.0, 4.1, and 5.0 (p. 1-16)	What's new and not-so-new in the Target Language Compiler
Where to Go from Here (p. 1-23)	Topics covered in this and related MATLAB® manuals

What Is the Target Language Compiler?

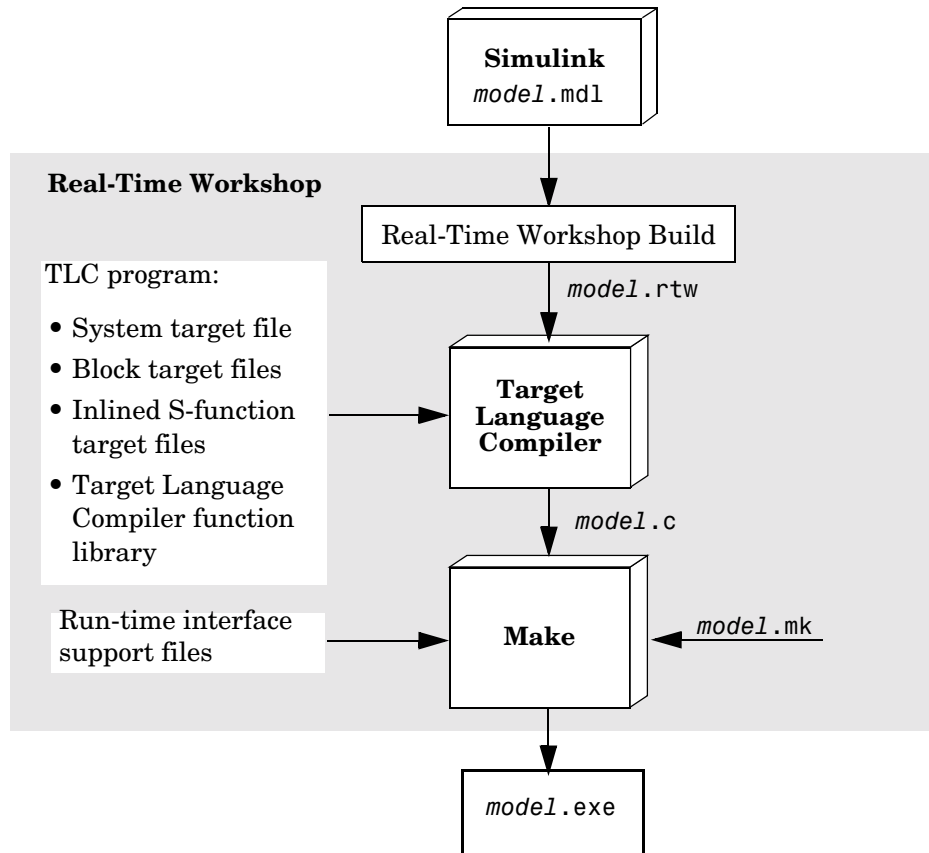
The Target Language Compiler is an integral part of the Real-Time Workshop[®]. It enables you to customize the C code generated from any Simulink[®] model and generate optimal, inlined code for your own Simulink blocks. Through customization, you can produce platform-specific code, or you can incorporate your own algorithmic changes for performance, code size, or compatibility with existing methods that you prefer to maintain.

Note This book describes the Target Language Compiler, its files, and how to use them together. This information is provided for those users who need to customize target files in order to generate specialized output or to inline S-functions in order to improve the performance and readability of the generated code. (The overall code generation process for the Real-Time Workshop is discussed in detail in “Code Generation and the Build Process” in the Real-Time Workshop documentation.)

This book refers to the Target Language Compiler either by its complete name, Target Language Compiler, or TLC.

Overview of the TLC Process

This top-level diagram shows how the Target Language Compiler fits in with the Real-Time Workshop 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.

As an integral component of Real-Time Workshop, the Target Language Compiler transforms an intermediate form of a Simulink block diagram, called `model.rtw`, into 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. The format of this file is described in “Contents of model.rtw” on page 4-1.

The word *target* in Target Language Compiler refers not only to the high-level language to be output, but 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 code.

In order to create a target-specific application, Real-Time Workshop also requires a template makefile that specifies the appropriate C compiler and compiler options for the build process. The template makefile is transformed into a makefile (*model.mk*) by performing token expansion specific to a given model. A target-specific version of the generic *rt_main* file (or *grt_main*) must also be modified to conform to the target’s specific requirements such as interrupt service routines. A complete description of the template makefiles and *rt_main* is included in the Real-Time Workshop documentation.

Those familiar with HTML, Perl, and MATLAB® will find that the Target Language Compiler borrows ideas from each of them. It has mark-up syntax similar to HTML, along with the power and flexibility of Perl and other scripting languages, plus the data handling power of MATLAB (TLC can invoke MATLAB functions). The code generated by TLC is highly optimized and fully commented C code, and can be generated from any Simulink model, including linear, nonlinear, continuous, discrete, or hybrid. All Simulink blocks are automatically converted to code, with the exception of MATLAB function blocks and S-function blocks that invoke M-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 “Inlining C MEX S-Functions” on page 7-5), thus improving performance by eliminating function calls to the S-function itself and the memory overhead of the S-function’s SimStruct. Inlining an S-function incorporates the S-function block’s code into the generated code for the model. When no TLC target file is present for the

S-function, its C code file is invoked via a function call. For more information on inlining S-functions, see “Inlining S-Functions” on page 7-1. You can also write target files for M-files or Fortran S-functions.

Overview of the Code Generation Process

Figure 1-1, The Target Language Compiler Process, on page 1-6 shows how the Target Language Compiler works with its target files and Real-Time Workshop output to produce code. When generating code from a Simulink model using Real-Time Workshop, the first step in the automated process is to generate a *model.rtw* file. The *model.rtw* file includes all of 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 directory. For all other files created during code generation, including the *model.rtw* file, a build directory is used. This directory is created by Real-Time Workshop right in the current directory and is named *.model_target_rtw*, where *target* is the abbreviation for the target environment, e.g., *grt* is the abbreviation for the generic real-time target.

As of Release 13 (version 5), files placed in the build directory include

- The body for the generated C source code (*model.c*)
- 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.

Note that in previous releases, generated source files were packaged as follows:

- The body for the generated C source code (*model.c*)
- Header files (*model.h* and *model_export.h*)
- A model registration include file (*model_reg.h*) that registers the model's `SimStruct`, sets up allocated data, and initializes nonfinite parameters
- A parameter include file (*model_prm.h*) that has information about all the parameters contained in the model

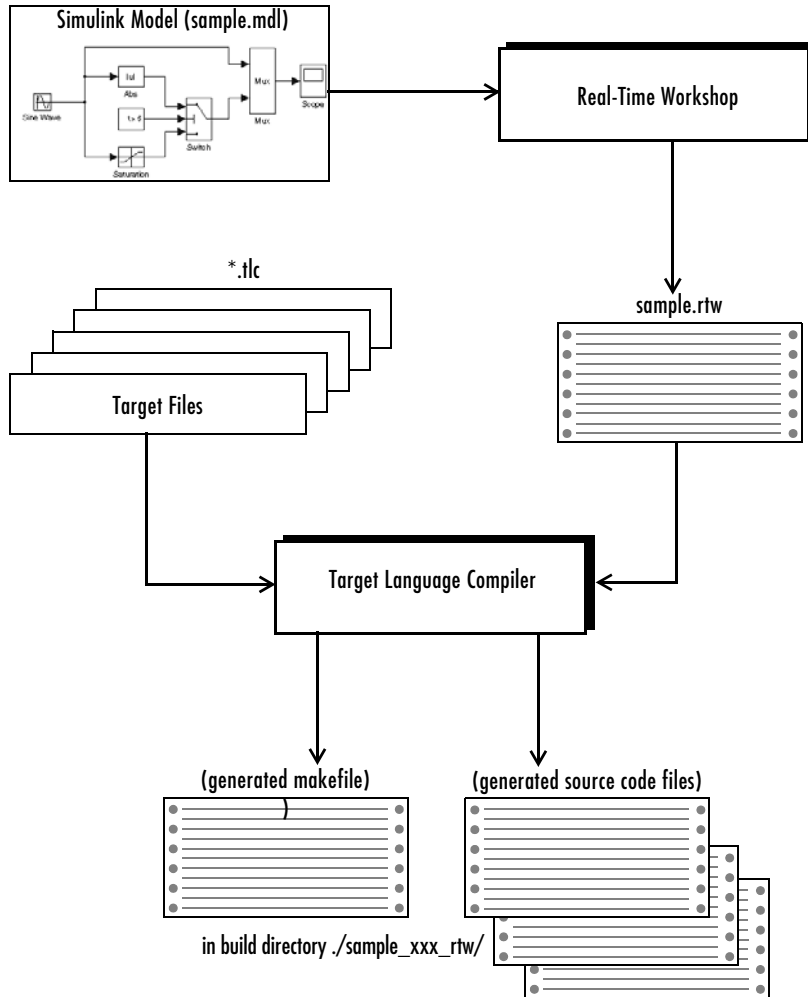


Figure 1-1: The Target Language Compiler Process

Target Language Compiler Capabilities

If you simply need to produce ANSI 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 of Real-Time Workshop, you will need to 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 Embedded MATLAB Function Block and Real-Time Workshop Embedded Coder facilitate code customization in a variety of ways. You may 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 datatypes 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” on page 5-1 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 3-18 for more information about target files.

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 may write to add your own algorithms, device drivers, and custom blocks to a Simulink model.

To create an S-function, you write C 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 all 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 and reduces memory usage of the generated code.

Modifying and Diversifying Code Generation

In principle, you can use the Target Language Compiler to convert the `model.rtw` file into any 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. The MathWorks supports, but does not recommend, doing this. In order for you to maintain such customizations, you may need to update your TLC files with each release of the Real-Time Workshop. The MathWorks continues to improve code generation by adding features and improving its efficiency, and possibly by altering the contents of the `model.rtw` file. We try to make such changes backwards compatible, but cannot guarantee that they all will be. However, inlined TLC files that users prepare are generally backwards compatible, provided that they invoke only documented TLC library and built-in functions.

Code Generation Process

Real-Time Workshop invokes the Target Language Compiler after a Simulink model is compiled into an intermediate form (*model.rtw*) that is suitable for generating code. To generate code appropriately, 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 M-file 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.

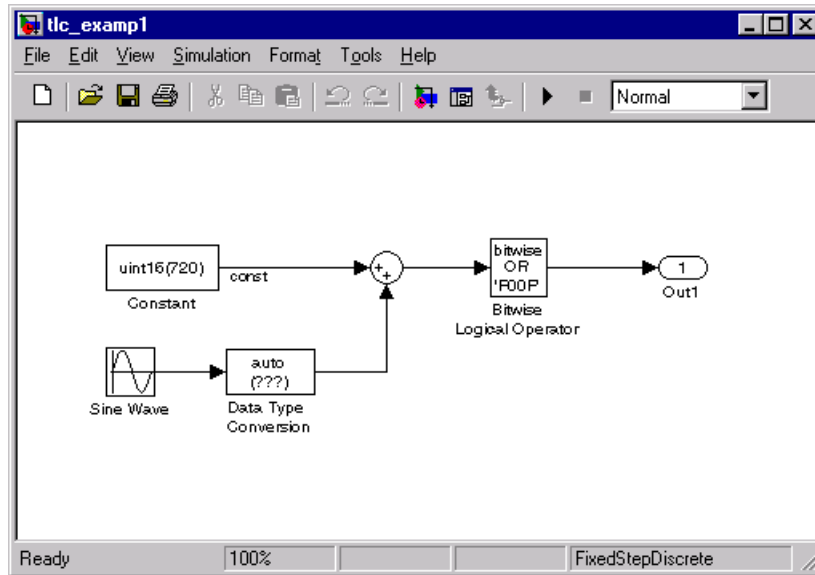
Because they cannot use SimStructs, Fortran and M-file S-functions must be inlined. This inlining can either be in the form of a full block target file or a “one-liner” block target file that references a “substitute” C MEX S-function source file.

A C MEX S-function will be selected for inlining by the Target Language Compiler if there is an explicit `mdlRTW()` function in the S-function code or if there is a target file for the current target language for the current block 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 will result.

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 example C MEX S-function source file `sfun_bitop.c` has its compiled form in `toolbox/simulink/blocks/sfun_bitop.dll` (`.mex*` for UNIX) and its C target file is located in `toolbox/simulink/blocks/tlc_c/sfun_bitop.tlc`.

A Look at Inlined and Noninlined S-Function Code

This example focuses on the example S-function `sfun_bitop.c` in directory `matlabroot/simulink/src/`. 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 bitwise 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 'FOOF' */
rtb_temp2 |= 0xF00F;
```

There is no initialization or setup code required for this inlined block.

If this block were noninlined, the source code for the S-function itself with all 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's 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 (sfun_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 (sfun_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 all 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 give handcoded size and performance to the generated code.

Advantages of Inlining S-Functions

Motivations

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 Inlining Is Not Appropriate

You may decide that inlining is not appropriate for certain C MEX S-functions. This may be the case if an S-function has

- Few or no numerical parameters
- One algorithm that is already fixed in capability (i.e., 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 may not improve execution speed and could actually increase the size of the generated code. The trade-off is in the size of the block's body code generated for each instance vs. 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 is used to simply generate a call to a custom code function that the S-function itself also calls. This approach may be the optimal solution for code generation in the case of a large piece of

existing code. An adaptation of this hybrid technique is used for calling the `rt_*.c` library functions located in directory `rtw/c/libsrc/`. See “Inlining S-Functions” on page 7-1 for the procedure and an example of a wrapped S-function.

Inlining Process

The strategy for achieving compact, high performance code from Simulink blocks in Real-Time Workshop centers on determining what part of a block’s operations are active and necessary 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, all 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 parameter set for each block is output in the initial phase of the code generation process from the S-function’s registered 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 Target Files

The Target Language Compiler has the following search path for block target files:

- 1 The current directory
- 2 The directory where the S-function executable (MEX or .m) file is located
- 3 S-function directory's subdirectory ./t1c_c (for C language targets)

The first target file encountered with the required name that implements the proper language will be used in processing the S-function's *model.rtw* file entry.

Availability for Inlining and Noninlining

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

Table 1-1: Inline TLC Support by S-Function Type

S-Function Type	Noninlining Supported	Inlining Supported
M-file	No	Yes
Fortran MEX	No	Yes
C	Yes	Yes

New Features and Compatibility Issues in Versions 4.0, 4.1, and 5.0

New Features Added in Version 5.0

The following features have been added to the Target Language Compiler for Version 5.0 (Release 13). In addition, Real-Time Workshop 5.0 contains many fixes and enhancements that are potentially relevant to Target Language Compiler users. See the Real-Time Workshop Release Notes documentation for a complete description. The Target Language Compiler 5.0 updates are

- A C-like `SPRINTF` built-in formatting function has been added, which returns a TLC string encoded with data from a variable number of arguments.
- `BlockInstanceData` function has been deprecated.
S-functions should no longer call the `BlockInstanceData` function. All data used by a block should be declared using data type work vectors (`DWORK`).
- Unified code generation for Real-Time Workshop and Stateflow®
In earlier releases, code generated from Stateflow charts in a model was written to source code files distinct from the source code files (such as `model.c`, `model.h`, etc.) generated from the rest of the model. Now, by default, Stateflow no longer generates any separate files from the Real-Time Workshop. For example, all Stateflow initialization code is now inlined.
- A new directive, `%filescope`, can be used to limit the scopes of variables to the files they are defined in. All variables defined after the appearance of `%filescope` in a file will have this property, otherwise they will default to global variables.
- Use of the `::` operator to access global variables is now allowed in TLC files. Variables defined on the command line and records read from `model.rtw` files will remain global variables. Nested include files cannot access variables local to the file which included them.
- The `%assert` directive (which tests the value of a Boolean expression and issues an error message, prints a stack trace, and then exits if the result is `FALSE`) is now easier to control.

You may enable/disable such `%assert` tests in several ways: via the `-da` command line switch, by the `%setcommandswitch` directive within files, using the `set_param(model, 'TLCAssertion', 'on|off')` command, or with a

check box control on the Real-Time Workshop GUI. By default, the check box is empty (%assert directives are ignored).

- The EXISTS builtin will now be able to take a nonstring expression as an argument. The old version of EXISTS will be deprecated (and will possibly generate a warning). The new EXISTS variation will be much faster than the old version.
- You may now request HTML reports when generating code for most targets (all except the S-Function target and the Rapid Simulation target).

New Features Added in Version 4.1

The following features have been added to the Target Language Compiler for Version 4.1 (Release 12.1):

- The TLC Debugger is now supported. See “Debugging TLC Files” on page 6-1
- ISINF, ISNAN, and ISFINITE now work for complex values.
- Added support for literal strings.

If a string constant is preceded by an L format specifier (as in L"string"), Target Language Compiler performs no escape character processing on that string. This is useful for specifying PC-style paths without using double backslash characters.

The following examples are equivalent.

- L"d:\this\is\a\path"
- "d:\\this\\is\\a\\path"

- Zero indexing for complex values is now supported, as in

```
%assign a = 1.0 + 3.0i
%assign b = a[0] %% this didn't work before
```

- The following new functions have been added to the TLC function library:
 - LibBlockInputSignalConnected
 - LibBlockInputSignalLocalSampleTimeIndex
 - LibBlockInputSignalOffsetTime
 - LibBlockInputSignalSampleTime
 - LibBlockInputSignalSampleTimeIndex
 - LibBlockOutputSignalOffsetTime
 - LibBlockOutputSignalSampleTime
 - LibBlockOutputSignalSampleTimeIndex

- LibBlockMatrixParameterBaseAddr
- LibBlockParameterBaseAddr
- LibBlockNonSampledZC

See “Inlining S-Functions” on page 7-1 for information on these functions.

- The handling of signal connections in `rtw/c/tlc/blkiolib.tlc` was reworked along with updating the help for `LibBlockInputSignal`. See “Input Signal Functions” on page 8-9.

New Features Added in Version 4.0

The following features were added to the Target Language Compiler for Version 4.0 (Release 12):

- Complete parsing of the TLC file just before execution. This aids development because syntax errors are caught the first time the TLC file is run instead of the first time the offending line is reached.
- TLC speed improvements across the board, particularly in block parameter generation
- Creation and use of a build directory in the current directory to prevent generated code from clashing with other files generated for other targets, and for keeping your model directories maintenance to a minimum
- Entirely new TLC Profiler for finding performance problems in your TLC code
- New format and changes to the `model.rtw` file.
- Aliases added for block parameters in the `model.rtw` file.
- New flexible methods for text expansion from within strings
- Column-major ordering of two-dimensional signal and parameter data
- `FIELDNAMES`, `GENERATE_FORMATTED_VALUE`, `GETFIELD`, `ISALIAS`, `ISEMPTY`, `ISEQUAL`, `ISFIELD`, `REMOVEFIELD`, `SETFIELD`. Support for two-dimensional signals in inlined code.
- `INTMAX`, `INTMIN`, `TLC_TRUE`, `TLC_FALSE`, `UINTMAX`
- Functions can return records.
- Formalization of records and record aliases
- Loop control variables are local to loop bodies.

- Improved EXISTS semantics; see “Built-In Functions and Values” on page 5-39
- Can expand records with %<>
- Short circuiting of conditionals (||, &&, ?:, %if-%elseif-%else-%endif)
- Relational operators can be used with nonfinite values.
- Enhanced conversion rules for FEVAL. You can now pass records and structs to FEVAL.

Compatibility Issues

Compatibility Issues in Version 5.0

In bringing Target Language Compiler files from Release 12.1 to Release 13, the following changes may affect your TLC code base:

- The BlockInstanceData function, as mentioned above, has been deprecated. Any TLC files that reference it should be updated.
- By default, GRT targets now use the rtModel data structure in place of the root SimStruct.

Designed to reduce code size and improve readability, the rtModel is a lightweight structure that is dynamically created when compiling a model, containing only those fields required to execute that model. GRT now utilizes the SimStruct only for noninlined child S-functions.

- Changes to the format of the *model.rtw* file may require you to update TLC files that access *model.rtw* records, especially if they do so directly rather than by calls to the TLC function library.

Compatibility Issues in Version 4.1

In bringing Target Language Compiler files from Release 12 to Release 12.1, the following changes may affect your TLC code base:

- The formats and default values for several important record structures in the *model.rtw* file have been changed. See “model.rtw Changes Between Real-Time Workshop 5.0 and 4.1” on page A-12 for further information.
- During the initialization phase of code generation, the order in which the Target Language Compiler calls each block’s BlockTypeSetup and

BlockInstanceSetup functions is different. In version 4.1, the BlockTypeSetup function is called before the BlockInstanceSetup function.

- The code generation variables FunctionInlineType and PragmaInlineString are now obsolete.

Compatibility Issues in Version 4.0

In bringing Target Language Compiler files from Release 11 to Release 12, the following changes may affect your TLC code base:

- Nested evaluations are no longer supported. Expressions such as `%<LibBlockParameter(%<myVariable>," ", " ", " ")>` are no longer supported. You will have to convert these expressions into equivalent nonnested expressions.
- Aliases are no longer automatically created for Parameter blocks while reading in Real-Time Workshop files.
- You cannot change the contents of a “Default” record after it has been created. In the previous TLC, you could change a “Default” record and see the change in all the records that inherited from that default record.
- `%codeblock` and `%endcodeblock` constructs are no longer supported.
- `%defines` & macro constructs are no longer supported.
- Use of line continuation characters (`. . .` and `\`) are not allowed inside of strings. Also, to place a double quote (`"`) character inside a string, you must use `\`. Previously, TLC allowed you to do `" "` to get a double quote in a string.
- Semantics have been formalized to `%include` files in different contexts (e.g., from generated files inside of `%with` blocks, etc.) `%include` statements are now treated as if they were read in from the global scope.
- The previous TLC had the ability to split function definitions (and other directives) across include file boundaries (e.g., you could start a `%function` in one file and `%include` a file that had the `%endfunction`). This no longer works.

- Nested functions are no longer allowed. For example:

```
%function foo ()
  %function bar ()
  %endfunction
%endfunction
```

- Recursive records are no longer allowed. For example:

```
Record1    {
  Val      2
  Ref      Record2
}
Record2    {
  Val      3
  Ref      Record1
}
```

- Record declaration syntax has changed. The following code fragments illustrate the differences between declaring a record `recVar` in previous versions of the Target Language Compiler and the current release.

- Previous versions:

```
%assign recVarAlias = recVar { ...
  field1 value1 ...
  field2 value2 ...
  ...
  fieldN valueN ...
}
```

- Current version:

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

See “Records” on page 3-14 for further information.

- Semantics of the EXISTS function have changed. In the previous release of Target Language Compiler, EXISTS(var) would check if the variable represented by the string value in var existed. In the current release of Target Language Compiler, EXISTS(var) checks to see if var exists or not.

To emulate the behavior of EXISTS in the previous release, replace

```
EXISTS(var)
```

with

```
EXISTS("%<var>")
```

Where to Go from Here

The remainder of this book contains both explanatory and reference material for the Target Language Compiler:

- “Getting Started” on page 2-1 describes the process that the Target Language Compiler uses to generate code, and general inlining S-function concepts.
- “Code Generation Architecture” on page 3-1 describes the TLC files and the build process. It also provides a tutorial on how to write target language files.
- “Contents of model.rtw” on page 4-1 describes the `model.rtw` file.
- “Directives and Built-in Functions” on page 5-1 contains the language syntax for the Target Language Compiler.
- “Debugging TLC Files” on page 6-1 explains how to use the TLC debugger.
- “Inlining S-Functions” on page 7-1 describes how to use the Target Language Compiler and how to inline S-functions.
- “TLC Function Library Reference” on page 8-1 contains abstracts for the TLC functions.
- “TLC Error Handling” on page A-1 lists the error messages that the Target Language Compiler can generate, as well as how to best use the errors.
- “Using TLC with Emacs” on page B-1 is a reference for using Emacs to edit TLC files.

Related Manuals

The items listed below are sections of other manuals that relate to the creation of TLC files:

- The Real-Time Workshop documentation describes the use and internal architecture of the Real-Time Workshop. The “Code Generation and the Build Process” chapter presents information on how Target Language Compiler fits into the overall code generation process. The “Targeting Real-Time Systems” chapter offers further useful examples and customization guidelines.
- The Real-Time Workshop Embedded Coder documentation presents details on generating code for embedded targets. Among other topics, it covers data structures and program execution, code generation, custom storage classes,

module packaging, and specifies system requirements and restrictions on target files.

- The Simulink Writing S-Functions documentation presents detailed information on all aspects of writing Fortran, M-file and C MEX S-functions. The most pertinent chapter from the point of view of the Target Language Compiler is “Guidelines for Writing C MEX S-Functions,” which details how to write wrapped and fully inlined S-functions with a special emphasis on the `mdlRTW()` function.

Getting Started

Code Architecture (p. 2-2)

What information code for a block captures

`model.rtw` and Target Language
Compiler Overview (p. 2-4)

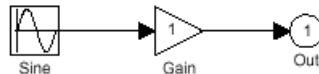
How the Target Language Compiler interprets `model.rtw`
files

Inlining S-Function Concepts (p. 2-6)

Techniques used for inlining, with examples

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 figure below, you see that blocks place code into Md1 routines. This shows Md1Outputs.



```

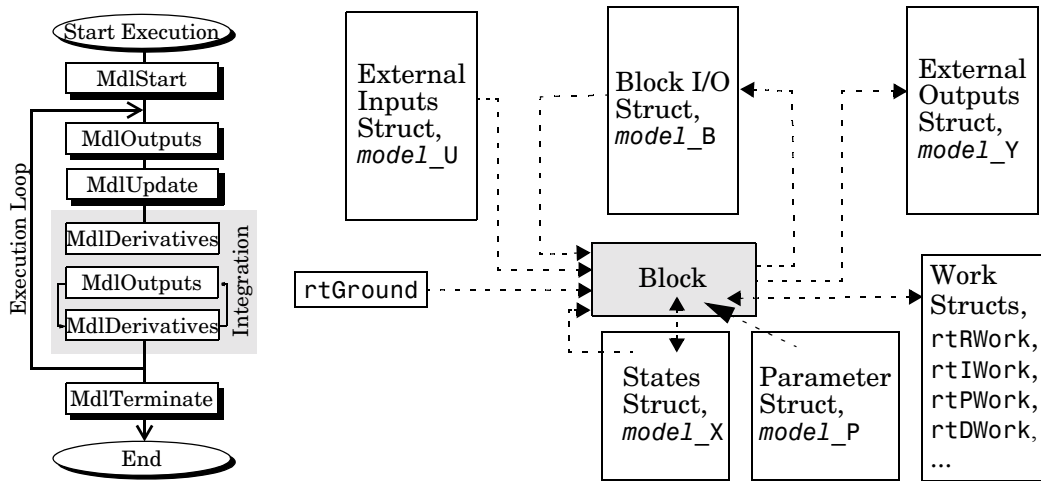
static void simple_output(int_T tid)
{
    /* Sin Block: '<Root>/Sine Wave' */

    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;

    /* Output: '<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 the Target Language Compiler-specific pieces of the code generation process.

model.rtw and Target Language Compiler Overview

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, Simulink 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 are comprised of 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, e.g., `grt.tlc`. This is the entry point to all the system TLC files as well as the block files, i.e., other TLC files get included into or generated from the one 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 all 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), i.e., 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 affect the generated code as desired.

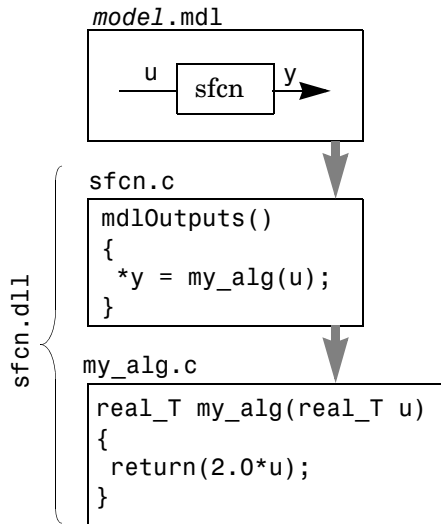
Inlining S-Function Concepts

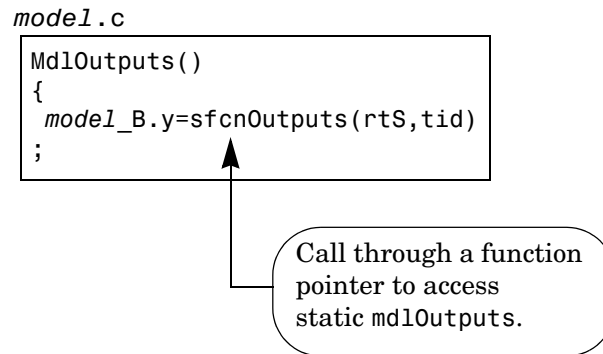
To inline an S-function means to provide a TLC file for an S-Function block that will replace the C (or Fortran or M) code version of the block that was used during simulation.

Noninlined S-Function

If an inlining TLC file is not provided, most Real-Time Workshop targets will still 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 the C coded S-function and only a limited subset of `mx*` API calls are supported within the Real-Time Workshop context. If you want the most efficient generated code, you must inline S-functions by writing a TLC file for them.

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





Types of Inlining

When inlining an S-function with a TLC file, it is helpful to define two categories of inlining:

- Fully inlined S-functions
- Wrapper inlined S-functions

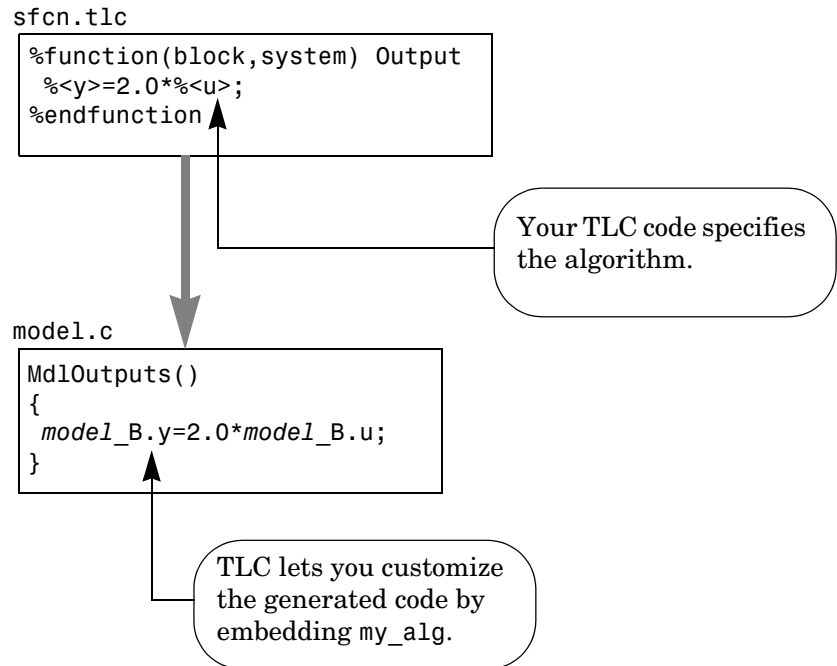
While both effectively inline the S-function and remove the overhead of a noninlined S-function, the two approaches are different. The first example below using `timetwo.tlc` is considered a fully inlined TLC file, where the full implementation of the block is contained in the TLC file for the block.

The second example uses a wrapper TLC file. Instead of generating all 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 of sharing the C code by both the C-MEX S-function and the generated code. There is no need to write the code twice.
- The called C function is an optimized routine.
- Several of the blocks may 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 the Real-Time Workshop 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 diagram below 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/simulink/src/timestwo.c` and the inlining TLC file for the block is in `matlabroot/toolbox/simulink/blocks/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` line is required by all TLC blocks file and is used by the Target Language Compiler to verify correct block type and correct language support 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 to 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, any nondirective lines in the `Outputs` function become generated code for the block.

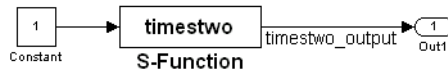
The most complicated piece of this TLC block example is the `%roll` directive. TLC uses this directive to provide for the automatic generation of for loops depending on input/output widths and whether the inputs are contiguous in memory. This example uses the typical form of accessing outputs and inputs 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 signal width.

The only function needed to implement this block is `Outputs`. For more complicated blocks, other functions will be declared as well. You can find examples of more complicated inlining TLC files in

`matlabroot/toolbox/simulink/blocks` 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.



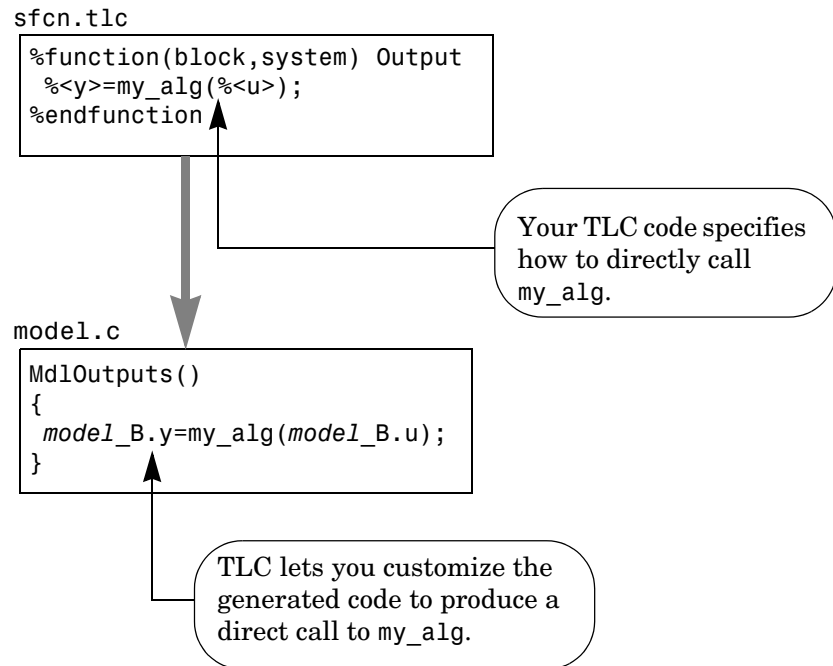
Model Outputs Code

```
/* 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;

    /* Output: '<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 models 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 multiply directly, the `Outputs` function now calls the function `mytimestwo`. So, all instances of this block in the model will call the same function to perform the multiply. 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);

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

Summary

This section has been a brief introduction to the `model.rtw` file and the concepts of inlining an S-function using the Target Language Compiler. "Contents of `model.rtw`" on page 4-1 contains more details of the `model.rtw` file and its contents. "Inlining S-Functions" on page 7-1 also contains details on writing TLC files.

Code Generation Architecture

Build Process (p. 3-2)	How the Target Language Compiler processes compiled model files to produce code
Invoking Code Generation (p. 3-8)	Running <code>rtwgen</code> and <code>t1c</code> from the MATLAB command line
Configuring TLC (p. 3-10)	How to pass in configuration data to customize builds
Code Generation Concepts (p. 3-13)	Understanding TLC variables and file and record handling
TLC Files (p. 3-18)	The roles and varieties of system and block target files
Data Handling with TLC: An Example (p. 3-26)	One way TLC library functions can transform data into data structures

Build Process

As part of the code generation process, Real-Time Workshop generates a *model.rtw* file from the Simulink model. This file contains information about the model that is then used to generate code. The code is generated through calls to a utility called the Target Language Compiler. The Target Language Compiler then converts these files into the desired language (e.g., C) and enables the code generation.

This section presents an overview of the build process, focusing more on the Target Language Compiler's role in this process.

The Target Language Compiler is a separate binary program that is included as a MEX-file. The Compiler compiles files written in the target language. The target language is an interpreted language, and thus, 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 the Target Language Compiler binary or any other such large binary to see the effects of your change.

Because the target language is an interpreted language, some statements may never be compiled or executed (and hence not checked by the compiler for correctness).

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

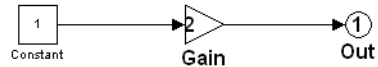
In the above example, the `Invalid_function_call` statement will never be executed. This example emphasizes that you should test all your the Target Language Compiler code with test cases that exercise every line.

A Basic Example

This section presents a basic example of creating a target language file that generates specific text from a Real-Time Workshop model. This example shows the sequence of steps that you should follow in creating and using your own target language files.

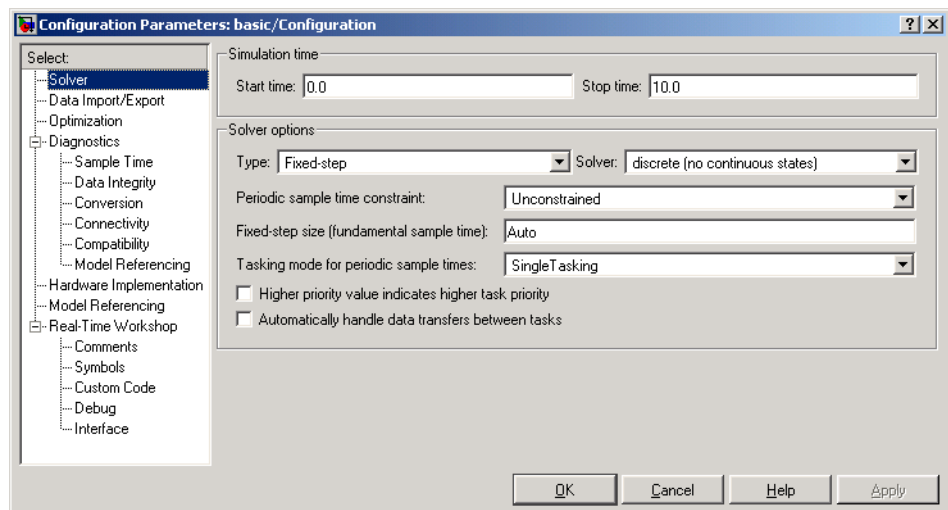
Process

To begin, create the Simulink model shown below and save it as `basic.mdl`.



Simulink Model

- 1 Select **Configuration Parameters** from the Simulink **Simulation** menu. This displays the **Configuration Parameters** dialog box
- 2 Select **Fixed-step** from the Solver pane of the **Configuration Parameters** dialog box.
- 3 Select **discrete (no continuous states)** from the **Solver** menu.
- 4 Click **Apply**. The dialog appears as below:



Configuration Parameters Dialog Box

- 5 Click **Real-Time Workshop** in the **Select** column to bring up the top-level Real-Time Workshop pane.
- 6 Click the **Generate Code only** button and then click **Apply**.
- 7 Click **Debug** in the **Category** column to activate the debug pane.
- 8 Select the **Retain .rtw file** option. This will let you inspect the contents of the *model.rtw* file after the build finishes.
- 9 Again, click **Real-Time Workshop** in the **Select** column to bring up the top-level Real-Time Workshop pane.
- 10 Click **Generate code**.

The build process then generates the code into the `basic_grt_rtw` directory and you can see the progress in the MATLAB window.

The output eventually displays

```
### Successful completion of Real-Time Workshop build procedure for model: basic
```

Viewing the `model.rtw` file `basic.rtw`

Open the file `./basic_grt_rtw/basic.rtw` in a text editor to see what it looks like. The hierarchy of records it contains includes among others, the following

elements (where elided lines are denoted by "..."; comments are delimited by < > and do not appear in the file).

```

CompiledModel {
<general model information, such as>
Name "basic"
Version "6.0 (R14 Prerelease 2) 06-Apr-2004"
ModelVersion "1.6"
GeneratedOn "Tue Apr 13 09:50:20 2004"
ExprFolding 1
TargetStyle StandAloneTarget
ModelReferenceTargetType "NONE"
AllowNoArgFcnInReusedFcn 0
PadderActive 0
PrmModelName SLDataModelName(basic)
TrigSSSplitOutUpd 1
UniqueFromFiles []
UniqueToFiles []

<Configuration set data starts here>
ConfigSet {
BlockReduction 0
BooleanDataType 0
BufferReuse 1
...
}
<Solver settings>
Solver FixedStepDiscrete
SolverType FixedStep
StartTime 0.0
StopTime 10.0
LoadInitialState no
...
}
<Global model settings>
NumModelInputs 0
NumModelOutputs 1
NumNonVirtBlocksInModel 3
DirectFeedthrough no
NumContStates 0
...
<Information specifying datatypes>
DataTypes {
NumDataTypes 14
NumSLBuiltInDataTypes 9
StrictBooleanCheckEnabled 0
DataType {
DTName double
Id 0
Size 8
...
}
}

```

```

    }
<External input specifications>
  ExternalInputs {
    ExternalInputDefaults {
      RecordType      ExternalInput
      Width           1
      MemoryMapIdx    [-1,-1,-1]
      HasObject       0
      DataTypeIdx     0
      ComplexSignal   no
      ...
    }
  }
  ...
<External ouyputs specifications>
  ExternalOutputs {
    ExternalOutputDefaults {
      RecordType      ExternalOutput
      Width           1
      MemoryMapIdx    [-1,-1,-1]
      SigLabel        ""
      HasObject       0
      Padding         0
    }
    NumExternalOutputs 1
    ExternalOutput {
      Block           [0, 2]
    }
  }
  BlockOutputs {
    GlobalBlockOutputDefaults {
      RecordType      BlockOutput
      SigSrc          []
      GrSrc           [-1, -1]
      ...
    }
  }
<Additional parameter records>
  ...
<Model checksum information>
  BlockParamChecksum Vector(4)
  ["1908524175U", "3510113275U", "2403630620U", "441379036U"]
  ModelChecksum Vector(4)
  ["2002754078U", "852865024U", "2143565807U", "1314203038U"]
  }

```

Creating the Target File

Next, create a basic .t1c file to act as a target file for this model. However, instead of generating code, simply print out 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 directory containing `basic.mdl`). 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 states.

%endwith
```

For the build process, you need to include some further information in the TLC file for the build process to successfully proceed. Instead, in this example, you will generate the `.rtw` file directly and then run the Target Language Compiler on this file to generate the desired output. To do this, enter at the MATLAB prompt

```
rtwgen('basic', 'OutputDirectory', 'basic_grt_rtw')
tlc -r basic_grt_rtw/basic.rtw basic.tlc -v
```

The first line generates the `.rtw` file in the build directory `'basic_grt_rtw'`, (this step is actually unnecessary since 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 the Target Language Compiler on the file `basic.tlc`. The `-r` option tells the Target Language Compiler 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 Mon Dec 03 09:42:13 2001.

It has 1 output(s) and 0 continuous states.
```

You may also try changing the model (such as 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.

Invoking Code Generation

Typically, `rtwgen` and TLC (as seen in the first section) are called directly from the Real-Time Workshop build procedure. This avoids problems that may arise due to command arguments changing from release to release. Thus you normally invoke `rtwgen` and `tlc` when you click the **Build (or Generate code)** button on the **Real-Time Workshop** dialog box. Sometimes, however, circumstances may require you to execute `rtwgen` and `tlc` directly from the MATLAB prompt.

The `rtwgen` Command

To generate the `model.rtw` file from the MATLAB prompt, it usually suffices to type

```
rtwgen('model')
```

However, you may want to specify a build directory in which to place the output file. You exercise this and other options using the keyword, value syntax.

```
rtwgen('model', 'OutputDirectory', '<build_directory>')
```

You may specify other options to `rtwgen`, such as whether or not identifiers should have case sensitivity, and reserved keywords. For more details, type

```
help rtwgen
```

at the MATLAB prompt.

The `tlc` Command

Once the `.rtw` file generates, to run the Target Language Compiler on this file, type

```
tlc -r build_directory/model.rtw file.tlc
```

This generates output as directed by `file.tlc`. Options to TLC include

- `-Ipath`, which specifies paths to look for files included by the `%<include>` directive (do not insert a space after `-I`)
- `-r model.rtw`, the compiled model file from which to generate code (note required space character before the argument)

- `-aident=expression`, which assigns a value to the TLC identifier `ident`. Note that there is *no* space after `-a`. Usage of `-a` is discussed in “Configuring TLC” on page 3-10.

For more details, type

```
help tlc
```

at the MATLAB prompt.

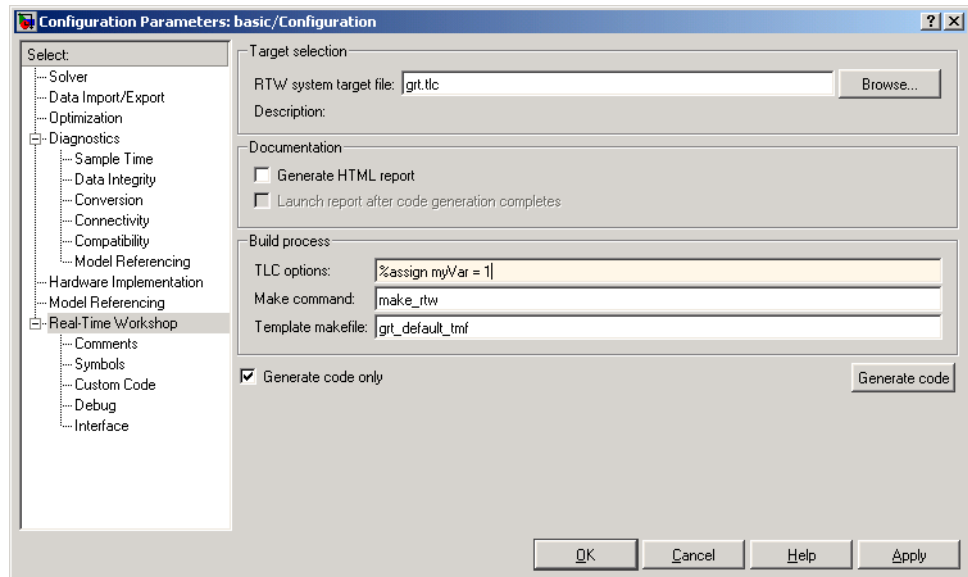
Configuring TLC

You can control and configure TLC in various ways, as the following sections explain.

Setting Command Line Arguments

You can enter TLC command line arguments from the MATLAB command line, or from the **TLC Options** text field on the **Real-Time Workshop** pane of the **Configuration Parameters** dialog box. This dialog is also accessible via **Tools -> Real-Time Workshop -> Options** on the Simulink menu bar.

You can enter commands in the TLC options field, as shown below.



The **TLC options** field turns yellow after you enter arguments. Click **Apply** to use the arguments you enter when the Target Language Compiler processes the model.

Another way of configuring the TLC code generation process is by using the `-a` flag on the TLC command line. That is you must give the TLC command interactively. Using `-amyVar=1` on the command line is equivalent to saying

```
%assign myVar = 1
```

in your target file, or entering it in the **TLC options** field, as shown above.

You can repeat the `-a` parameter, which also can be specified in the **System Target File** field in the Target Configuration section of the **Real-Time Workshop** dialog box.

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 error out if you forget to add `-amyConfigVariable` to the command line.

If you use the `-a` flag to input a string variable, the variable must be enclosed in double quotes:

```
-amyStringVariable="hello"
```

However, if the string contains any white space, enclose the double quotes within apostrophes:

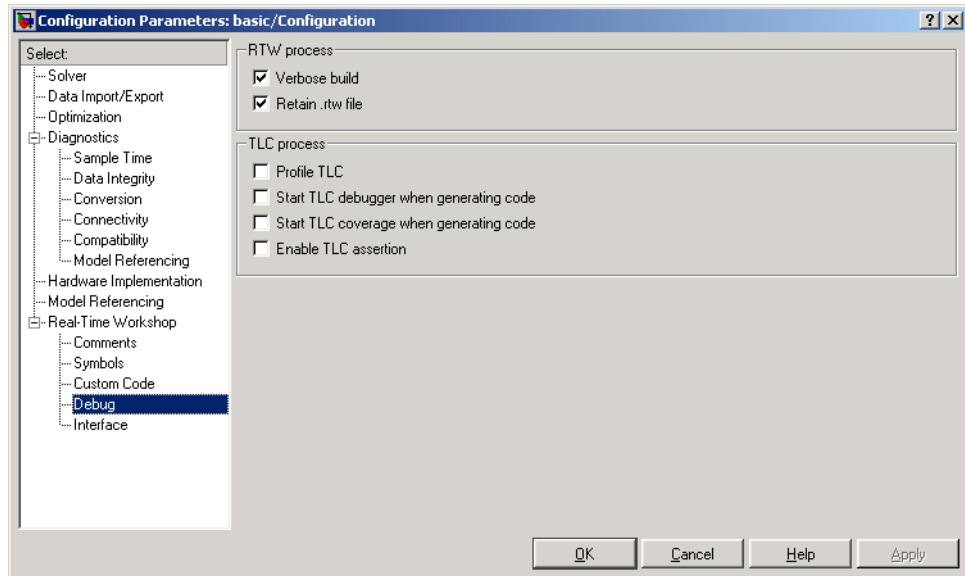
```
-amyStringVariable=' "hello world" '
```

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

```
-amyStringVariable=' "can' 't" '
```

Configuring for TLC Debugging

To configure TLC for debugging via the **Configuration Parameters** dialog, select **Debug** under **Real-Time Workshop**. 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 and an option to retain the RTW file. This is covered in more detail in “Debugging TLC Files” on page 6-1.

Code Generation Concepts

The Target Language Compiler uses a *target language* that is a general programming language, and you can use it as such. It is important, however, to remember that the Target Language Compiler 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 provide.

Before you start modifying or creating target files for use within the Real-Time Workshop, you might find some of the following general programming examples useful to familiarize yourself with the basic constructs used within the Target Language Compiler.

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 Target Language Compiler program, type

```
tlc hello.tlc
```

at the MATLAB prompt.

This simple script demonstrates some important concepts underlying the purpose (and hence the design) of the Target Language Compiler. Since the primary purpose of the Target Language Compiler 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 the Target Language Compiler to send any following text that it generates or does not recognize to the standard output device. All syntax that the Target Language Compiler recognizes begins with the `%` character. Since `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 in order 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 the Compiler Directives table.

Variable Types

The absence of explicit type declarations for variables is another feature of the Target Language Compiler. See “Directives and Built-in Functions” on page 5-1 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 variable that references this record while `recType` is the record itself. `fieldi` is a string and `valuei` is the corresponding Target Language Compiler 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. Thus, a simple script that loops through a model and outputs the name of all blocks in the model would have the following form.

```
%include "utllib.tlc"  
%selectfile STDOUT  
%with CompiledModel  
    %foreach sysIdx = NumNonvirtSubsystems + 1  
        %assign ss = System[sysIdx]  
        %with ss  
            %foreach blkIdx = NumBlocks  
                %assign block = Block[blkIdx]  
                %<LibGetFormattedBlockPath(block)>  
            %endforeach  
        %endwith  
    %endforeach  
%endwith
```

Unlike MATLAB, the Target Language Compiler requires that you explicitly load any function definitions not located in the same target file. In MATLAB, the line `A = myfunc(B)` causes MATLAB to automatically search for and load an M-file or MEX-file named `myfunc`. The Target Language Compiler, on the other hand, requires that you specifically include the file that defines the function. In this case, `utllib.tlc` contains the definition of `LibGetFormattedBlockPath`.

Target Language Compiler provides a `%with` directive that facilitates using records. See “Directives and Built-in Functions” on page 5-1 for a detailed description of the directive and its associated scoping rules.

Note The format and structure of the *model.rtw* file are subject to change from one release of Real-Time Workshop to another.

A record read from a file is not immutable. It is like any other record that you might declare in a program. In fact, the global `CompiledModel` Real-Time Workshop record is modified many times during code generation. `CompiledModel` is the global record in the *model.rtw* file. It contains all the variables necessary for code generation such as `NumNonvirtSubsystems`,

NumBlocks, etc. It is also appended during code generation with many new variables, flags, and subrecords as needed.

Functions such as LibGetFormattedBlockPath are provided in the Target Language Compiler libraries located in *matlabroot/rtw/c/tlc/lib/*.tlc*. For a complete list of available functions, refer to “TLC Function Library Reference” on page 8-1.

Assigning 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 which 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 = 2, %<a.foo.field>
b.foo.field = 2, %<b.foo.field>
c.foo.field = 1, %<c.foo.field>
%% note that c.foo.field is unchanged

```

It is possible to create aliases to records which are not attached to any other records, 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
  %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 command window output

```

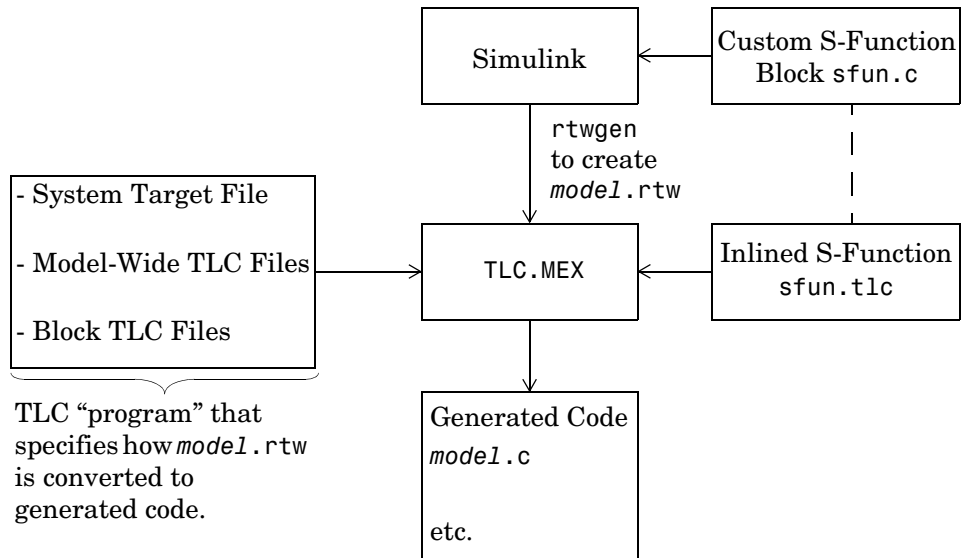
ISALIAS(a.foo) = 1
ISALIAS(x) = 1
x = { field 2 }
x.field = 2

```

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

TLC Files

The Target Language Compiler works with Simulink 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 also a collection of ASCII files, also called *scripts*. Since the Target Language Compiler is an interpreted language, however, there are no object files. The single target file that calls (with the `%include` directive) all other target files needed for the program is called the *entry point*.

Available Target Files

Target files are the set of files that are interpreted by the Target Language Compiler to transform the intermediate Real-Time Workshop 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. By modifying the target files included with the Compiler, you can dictate what the compiler produces. For example,

if you use the available system target files, you produce generic C code from your Simulink model. This executable C code is not platform specific.

All of 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. The block scoping rules and the `%assign` statement are discussed in “Directives and Built-in Functions” on page 5-1.

Target files are written using target language directives. “Directives and Built-in Functions” on page 5-1 provides complete descriptions of the target language directives.

Model-Wide Target Files and System Target Files

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

The system target file is the *entry point* for the Target Language Compiler. 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 Real-Time Workshop target files. For a complete list of available system target files for Real-Time Workshop, see the Real-Time Workshop documentation.

There are four sets of model-wide target files, one for each of the basic code formats that the Real-Time Workshop supports. The following table lists the model-wide target files associated with each of the basic code formats.

Table 3-1: Model-Wide Target Files for Static Real-Time, Malloc (dynamic) Real-Time, Embedded-C and Real-Time Workshop S-Function Applications

Model-Wide Target File	Code Format	Purpose
ertautobuild.tlc	Embedded-C	Includes <i>model_export.h</i> in the generated code
srtbody.tlc mrtbody.tlc ertbody.tlc sfcnbody.tlc	Static real-time Malloc real-time Embedded-C RTW S-function	Creates the source file, <i>model.c</i> , which contains the procedures that implement the model
srtexport.tlc mrtextport.tlc ertexport.tlc sfcnbody.tlc	Static real-time Malloc real-time Embedded-C RTW S-function	Creates the header file <i>model_export.h</i> , which defines access to external parameters and signals (all formats)
srthdr.tlc mrthdr.tlc erthdr.tlc sfcnhdr.tlc	Static real-time Malloc real-time Embedded-C RTW S-function	Creates the header file <i>model.h</i> , which defines the data structures used by <i>model.c</i> . The data structures defines include Block Outputs, Parameters, External Inputs and Outputs, and the various work structures. The instances of these structures are declared in <i>model.c</i> (all formats).
srtlib.tlc mrtlib.tlc ertlib.tlc sfcplib.tlc	Static real-time Malloc real-time Embedded-C RTW S-function	Contains utility functions used by the other model-wide target files (all formats)
srtmap.tlc mrtmap.tlc ertmap.tlc sfcnmap.tlc	Static real-time Malloc real-time Embedded-C RTW S-function	Creates the header file <i>model.dt</i> , which contains the mapping information for monitoring block outputs and modifying block parameters

Table 3-1: Model-Wide Target Files for Static Real-Time, Malloc (dynamic) Real-Time, Embedded-C and Real-Time Workshop S-Function Applications

Model-Wide Target File	Code Format	Purpose
sfcnmid.tlc	RTW S-function	Creates <i>model.c</i> , which contains data for an RTW S-function
srtparam.tlc mrtparam.tlc ertparam.tlc sfcnparam.tlc	Static real-time Malloc real-time Embedded-C RTW S-function	Creates the source file <i>model.prm</i> , which is included by the <i>model.c</i> file to declare instances of the various data structures defined in <i>model.h</i> (all formats)
srtreg.tlc mrtreg.tlc ertreg.tlc sfcnreg.tlc	Static real-time Malloc real-time Embedded-C RTW S-function	Creates the source file <i>model.h</i> that is included by the <i>model.c</i> file to satisfy the API (all formats)
sfcnsid.tlc	RTW S-function	Creates <i>model.c</i> , which contains data for an RTW S-function.
srtwide.tlc mrtwide.tlc ertwide.tlc sfcnwide.tlc	Static real-time Malloc real-time Embedded-C RTW S-function	The entry point for code format. This file produces <i>model.c</i> , <i>model.h</i> , and, optionally, <i>model.dt</i> .

Block Target Files

Block target files are files that control a particular Simulink block. Typically, there is a block target file for each Simulink basic building block. These files control the generation of inline code for the particular block type. For example, the target file, `gain.tlc`, generates corresponding code for the Gain block.

The file `genmap.tlc` (included by `codegenentry.tlc`) tells TLC which `.tlc` files to include for particular blocks.

Note Functions declared inside a block file are local. Functions declared in all other target files are global.

Summary of Target File Usage

In the context of the Real-Time Workshop, 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 get executed, how data gets logged, and so on.
- Block target files
Block target files determine how each individual block uses its input signals and/or parameters to generate its output or to update its state.

You must write or modify a target file if you need to do one of the following:

- Customize the code generated for a block

The code generated for each block is defined by a *block target file*. Some of the things defined in the block target file include what the block outputs at each major time step and what information the block updates.

- Inline an S-function

Inlining an S-function means writing a target file that tells the Target Language Compiler how to generate code for that S-function block. The Target Language 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 are executed using the S-function Application Program Interface (API) and can be inefficient.

It is possible to inline an M-file or Fortran S-function; the Target Language Compiler can generate code for the S-function in both these cases.

- Customize the code generated for all models

You may 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.

- Implement support for a new language

The Target Language Compiler provides the basic framework to configure the entire Real-Time Workshop for code generation in another language.

Refer to “Directives and Built-in Functions” on page 5-1 for a description of the Target Language and “Inlining S-Functions” on page 7-1 for a tutorial on using the Target Language Compiler to inline S-functions.

System Target Files

The entire code generation process starts with the single system target file that you specify in the **Real-Time Workshop** 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 MatFileLogging = 1
%assign TargetType = "RT"
%assign Language = "C"

%include "codegenentry.tlc"
```

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

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 code below 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"

%% Load mygenmap, the block target file mapping.
%% mygenmap.tlc should %include genmap.tlc at the beginning.
#include "mygenmap.tlc"

#include "commonsetup.tlc"

%% Next, you can include any of the TLC files that you need for
%% preprocessing information about the model and to fill in
%% Real-Time Workshop hooks. The following is an example of
%% including a single TLC file which 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 is detailed in the Real-Time Workshop documentation. During code generation, functions from each of the block target files are executed and the generated code is placed in the appropriate model or subsystem functions.

Block Target Files

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.

Block Target File Mapping

The *block target file mapping* specifies which target file should be used to generate code for which block type. This mapping resides in `matlabroot/rtw/c/tlc/mw/genmap.tlc`. All the TLC files listed are located in directories within `matlabroot/rtw/c/tlc`.

The Target Language Compiler works with various sets of script files to produce its results. The complete set of these files is called a *TLC program*. This section describes the TLC program files.

Data Handling with TLC: An Example

Matrix Parameters in Real-Time Workshop

MATLAB, Simulink, and Real-Time Workshop all use column-major ordering for all array storage (1-D, 2-D, ...), so that the “next” element of an array in memory is always accessed by incrementing the first index of the array. For example, all of 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 “The MATLAB Array” in External Interfaces/API.

Simulink and Real-Time Workshop differ from MATLAB internal data storage format only in the storage of complex number arrays. In MATLAB, the real and imaginary parts are stored in separate arrays, while in Simulink and Real-Time Workshop 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 a *.rtw* file and paste it into a *.m* file and have it recognized by MATLAB.

The Target Language Compiler declares all Simulink block matrix parameters as scalar or 1-D array variables

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

where *real_T* could actually be any of the data types supported by Simulink, and will match the variable type given in a the *.mdl* 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*

```

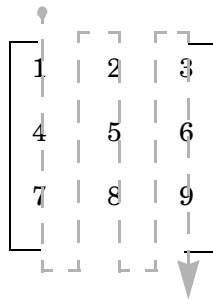
typedef struct Parameters_tag {
    real_T s1_Look_Up_Table_2_D_Table[9];
    /* Variable:s1_Look_Up_Table_2_D_Table
     * External Mode Tunable:yes
     * Referenced by block:
     * <S1>/Look-Up Table (2-D)
     */

    [ ... 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 ... ]  
};
```

The Target Language Compiler 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 any 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`.

Contents of model.rtw

The input to the Target Language Compiler is a *model.rtw* file, a compilation of *model.mdl* that describes blocks, inputs, outputs, parameters, states, storage, and other model components and properties.

Model.rtw File Overview (p. 4-2)	Identifiers, values, and the structure of records
Using Library Functions to Access model.rtw Contents (p. 4-10)	The safe way to access model records, with one exception

Note Please be aware that the structure of the *model.rtw* file is very likely to change between releases, which is a very compelling reason to limit your access to *model.rtw* to the TLC library functions documented in “TLC Function Library Reference” on page 8-1.

Model.rtw File Overview

Real-Time Workshop generates a *model.rtw* file from your Simulink model. The *model.rtw* file is a database whose contents provide a description of the individual blocks within the Simulink model. By selecting **Retain .rtw file** from the TLC debugging category on the Real-Time Workshop pane of the **Simulation Parameters** dialog box, you can build a model and view the corresponding *model.rtw* file that was used.

model.rtw is an ASCII file of parameter-value pairs stored in a hierarchy of records defined by your model. 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  
}
```

Using Scopes in the model.rtw File

Accessing Values

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 any value 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
  Block {                                   nonvirtual subsystem.
    Type    "S-Function"           - Block records for each
    Name    "<S3>/S-Function"       nonvirtual block in the system.
    ...
    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
```

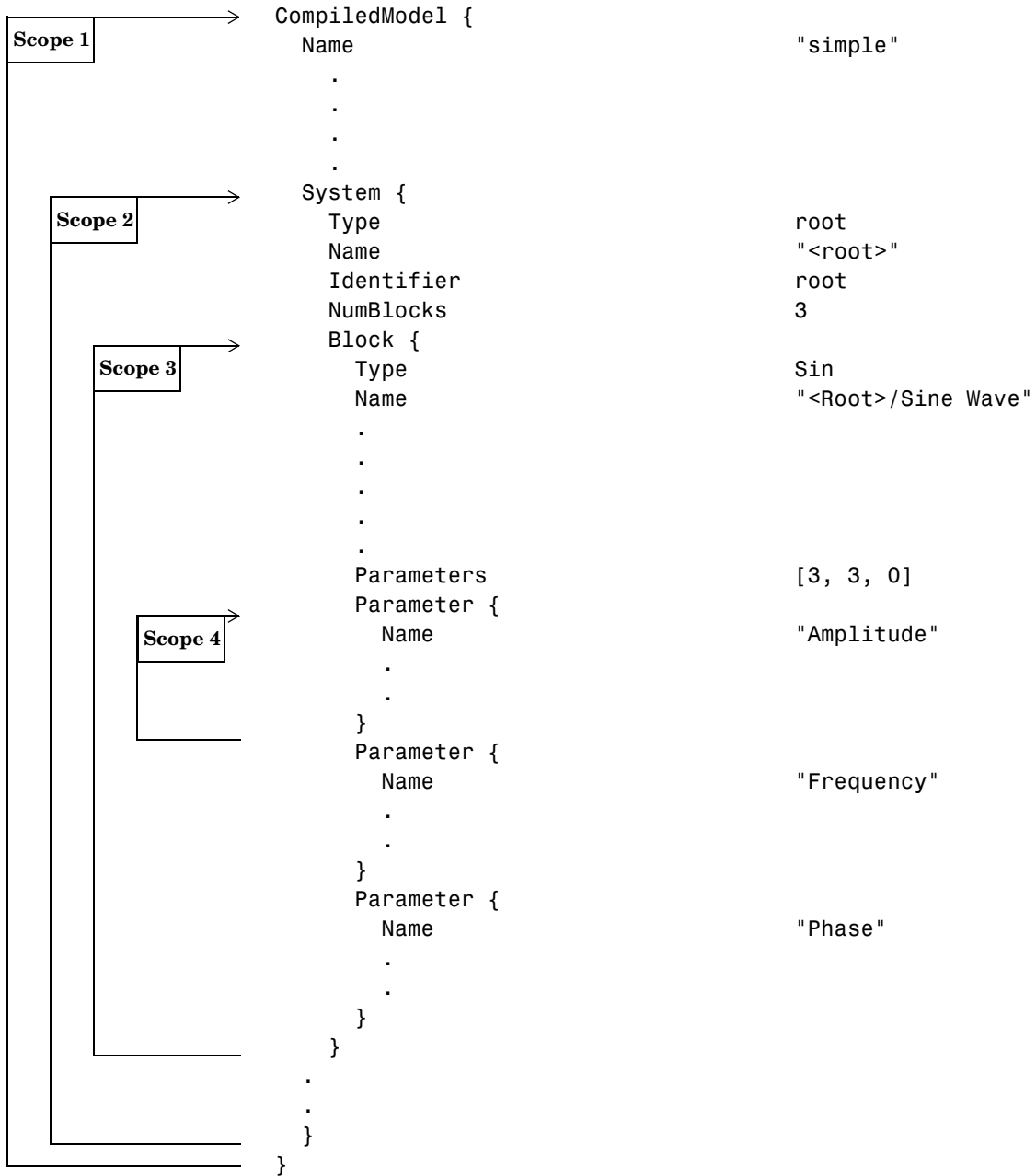
To simplify this process, you can use the `%with` directive, which changes the current scope. For example:

```
%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`.



Object Information in the model.rtw File

During code generation, Real-Time Workshop writes information about signal and parameter objects to the `model.rtw` file. An Object record is written for each parameter or signal that meets certain conditions. These conditions are described in “Object Records For Parameters” on page 4-6 and “Object Records For Signals” on page 4-7.

The Object records contain all of the information corresponding to the associated object. To access Object records, you must write Target Language Compiler code (see “Accessing Object Information via TLC” on page 4-8).

Object Records For Parameters

An Object record is included in the in the `ModelParameters` section of the `model.rtw` file for each parameter, under the following conditions:

- 1 The parameter resolves to a `Simulink.Parameter` object (or to a parameter object that comes from a class derived from the `Simulink.Parameter` class).
- 2 The parameter’s symbol is preserved in the generated code. The symbol is preserved when:
 - **Inline parameters** is on.
 - `RTWInfo.StorageClass` is not set to 'Auto' or 'SimulinkGlobal'.

The following is an example of an Object record for a parameter.

```
ModelParameters {
  ...
  Parameter {
    Identifier      Kp
    Tunable         yes
    ...
    Value           [5.0]
    Dimensions      [1, 1]
    HasObject       1
    Object {
      Package       Simulink
      Class          Parameter
      ObjectProperties {
        RTWInfo {
          Object {
```

```

Package           Simulink
Class             RTWInfo
ObjectProperties {
  StorageClass    "SimulinkGlobal"
}
}
}
Value            5.0
...
}
}
}
}
}
}

```

Object Records For Signals

An Object record is included in the BlockOutputs section of the *model.rtw* file for each signal which meets the following conditions:

- 1 The signal resolves to a `Simulink.Signal` object (or to an object that comes from a class derived from the `Simulink.Signal` class).
- 2 The signal's symbol is preserved in the generated code. The symbol is preserved if:
 - The signal's `RTWInfo.StorageClass` is not set to 'Auto' or 'SimulinkGlobal'.
 - The signal label is be a valid variable name.
 - The signal label is unique throughout the model.

Note 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 always preserved.

The following is an example of an Object record for a signal:

```

BlockOutputs {
  ...
  BlockOutput {

```

```

Identifier          SinSig
...
SigLabel "SinSig"
HasObject          1
Object {
  Package          Simulink
  Class            Signal
  ObjectProperties {
    RTWInfo {
      Object {
        Package          Simulink
        Class            RTWInfo
        ObjectProperties {
          StorageClass "SimulinkGlobal"
        }
      }
    }
  }
  ...
}
}
}
}

```

Accessing Object Information via TLC

This section provides sample code to illustrate how to access object information from the *model.rtw* file using TLC code. For more information on TLC and the *model.rtw* file, see “model.rtw” on page A-1.

Accessing Parameter Object Records. The following code fragment iterates over the ModelParameters section of the *model.rtw* file and extracts information from any 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 RTWInfo.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

```

Accessing Signal Object Records. The following code fragment iterates over the BlockOutputs section of the *model.rtw* file and extracts information from any signal object records encountered.

```

%with CompiledModel.BlockOutputs
%foreach blockOutputIdx = NumBlockOutputs
    %assign thisBlockOutput = BlockOutput[blockOutputIdx]
    %assign signalName = thisBlockOutput.Identifier
    %if EXISTS("thisBlockOutput.Object.ObjectProperties")
        %with thisBlockOutput.Object.ObjectProperties
            %with RTWInfo.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

```

Using Library Functions to Access model.rtw Contents

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 apposed to accessing the parameter name/parameter values pairs directly from your block TLC code.

See “TLC Function Library Reference” on page 8-1 for a list of the commonly used library functions.

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 may 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 instances 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 no longer work.
- TLC library functions are provided that substantially reduce the amount of TLC code needed to implement a block while handling all 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 ensures that it will be flexible enough to generate code for any instance or configuration of the block, as well as across releases. Exceptions to this do occur, however, as when it is necessary 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 4-11.

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 nonchanging values and information that is then used to affect 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” on page 7-1.

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 `t1c` command itself.

Compiler Directives (p. 5-2)

The syntax and formats of directives, built-in functions, signal and parameter values, expressions and comments

Command Line Arguments (p. 5-71)

Description of TLC calling arguments, filenames and search paths

Compiler Directives

Syntax

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

```
[text | %<expression>]* and
%keyword [argument1, argument2, ...]
```

Statements of the first type cause all 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 any 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 always 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.

The following table shows the complete set of Target Language Compiler directives. The remainder of this chapter describes each directive in detail.

Target Language Compiler Directives

Directive	Description
%% text	Single line comment where text is the comment
/% text %/	Single (or multi-line) comment where text is the comment
%matlab	Calls a MATLAB function that does not return a result. For example, %matlab disp(2.718)

Target Language Compiler Directives (Continued)

Directive	Description
%<expr>	<p>Target language expressions which are evaluated. For example, if we have a TLC variable that was created via: <code>%assign varName = "foo"</code>, then <code>%<varName></code> would expand to <code>foo</code>. Expressions can also be function calls as in <code>%<FcnName(param1,param2)></code>. On directive lines, TLC expressions do not need to be placed within the <code>%<></code> syntax. Doing so will cause a double evaluation. For example, <code>%if %<x> == 3</code> is processed by creating a hidden variable for the evaluated value of the variable <code>x</code>. The <code>%if</code> statement then evaluates this hidden variable and compares it against 3. The efficient way to do this operation is to do: <code>%if x == 3</code>. In MATLAB notation, this would equate to doing <code>if eval('x') == 3</code> as opposed to <code>if x = 3</code>. The exception to this is during a <code>%assign</code> for format control as in</p> <pre style="margin-left: 40px;">%assign str = "value is: %<var>"</pre> <p>Note: Nested evaluation expressions (e.g., <code>%<foo(%<expr>)></code>) are not supported.</p> <p>Note: There is no speed penalty for evals inside strings, such as</p> <pre style="margin-left: 40px;">%assign x = "%<expr>"</pre> <p>Evals outside of strings, such as the following example, should be avoided whenever possible.</p> <pre style="margin-left: 40px;">%assign x = %<expr></pre>

Target Language Compiler Directives (Continued)

Directive	Description
<pre>%if expr %elseif expr %else %endif</pre>	<p>Conditional inclusion, where the constant-expression <code>expr</code> must evaluate to an integer. For example, the following code checks whether a parameter, <code>k</code>, has the numeric value 0.0 by executing a TLC library function to check for equality.</p> <pre>%if ISEQUAL(k, 0.0) <text and directives to be processed if, k is 0.0> %endif</pre> <p>In this and other directives, it is not necessary to expand variables or expressions using the <code>%<expr></code> notation unless <code>expr</code> appears within a string. For example:</p> <pre>%if ISEQUAL(idx, "my_idx%<i>"), where idx and i are both strings.</pre> <p>As in other languages, logical evaluations do short circuit (are halted as soon as the result is known).</p>
<pre>%switch expr %case <i>expr</i> %break %default %break %endswitch</pre>	<p>The <code>switch</code> directive is very similar to the C language <code>switch</code> statement. The expression, <code>expr</code>, can be of any type that can be compared for equality using the <code>==</code> operator. If the <code>%break</code> is not included after a <code>%case</code> statement, then it will fall through to the next statement.</p>

Target Language Compiler Directives (Continued)

Directive	Description
<pre>%with %endwith</pre>	<p><code>%with recordName</code> is a scoping operator. Use it to bring the named record into the current scope, to remain until the matching <code>%endwith</code> is encountered (<code>%with</code> directives may be nested as desired).</p> <p>Note that on the left side of <code>%assign</code> statements contained within a <code>%with / %endwith</code> block, references to fields of records must be fully qualified (see “Assigning Values to Fields of Records” on page 3-16), as in the following example.</p> <pre>%with CompiledModel %assign oldName = name %assign CompiledModel.name = "newname" %endwith</pre>
<pre>%setcommandswitch string</pre>	<p>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</p> <p>The following example sets the verbosity level to 1.</p> <pre>%setcommandswitch "-v1"</pre> <p>See also “Command Line Arguments” on page 5-71.</p>
<pre>%assert expr</pre>	<p>Tests a value of a Boolean expression. If the expression evaluates to false TLC will issue an error message, a stack trace and exit, and otherwise the execution will be continued as normal. To enable the evaluation of asserts outside the Real-Time Workshop environment, use the command line option “-da”. When building from within RTW, this flag is not needed and will be ignored, as it is superseded by the Enable TLC Assertions check box on the TLC debugging section of the Real-Time Workshop pane. To control assertion handling from the MATLAB command window, use</p> <pre>set_param(model, 'TLCAssertion', 'on off')</pre> <p>to set this flag on or off. Default is Off.</p> <pre>get_param(model, 'TLCAssertion')</pre> <p>to see the current setting.</p>

Target Language Compiler Directives (Continued)

Directive	Description
<pre>%error %warning %trace %exit</pre>	<p>Flow control directives:</p> <pre>%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.</pre> <p>Note, when reporting errors, you should use</p> <pre>%exit Error Message</pre> <p>if the error is produced by an incorrect configuration that the user needs to correct in the model. If you are adding assert code (i.e., code that should never be reached), use</p> <pre>%setcommandswitch "-v1" %% force TLC stack trace %exit Assert message</pre>
<pre>%assign</pre>	<p>Creates identifiers (variables). The general form is</p> <pre>%assign [::]variable = expression</pre> <p>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).</p> <p>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</p> <pre>%assign nameInfo = "The name of this is %<Name>"</pre> <p>or alternately</p> <pre>%assign nameInfo = "The name of this is " + Name</pre> <p>To assign a value to a field of a record you must use a <i>qualified variable expression</i>. See “Assigning Values to Fields of Records” on page 3-16.</p>

Target Language Compiler Directives (Continued)

Directive	Description
%createrecord	<p data-bbox="454 343 1326 499">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.</p> <pre data-bbox="486 522 1285 612"> %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 */ </pre> <p data-bbox="454 638 1311 701">If more than one record specification follows a given record name, the set of record specifications constitutes an array of records.</p> <pre data-bbox="486 730 1285 881"> %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 */ </pre> <p data-bbox="454 907 1326 970">Note that arrays of subrecords can be created and indexed by specifying %createrecord with identically named subrecords, as follows:</p> <pre data-bbox="486 994 1285 1145"> %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 */ </pre> <p data-bbox="454 1171 1225 1234">If the scope resolution operator (::) is the first token after the %createrecord token, the record is created in the global scope.</p>

Target Language Compiler Directives (Continued)

Directive	Description
%addtorecord	<p>Adds fields to an existing record. The new fields may be name-value pairs or aliases to already existing records.</p> <pre data-bbox="491 427 906 453">%addtorecord OLD_RECORD foo 1</pre> <p>If the new field being added is a record, then %addtorecord will make an alias to that record instead of a deep copy. To make a deep copy, use %copyrecord.</p> <pre data-bbox="491 595 1228 652">%createrecord NEW_RECORD { foo 1 } %addtorecord OLD_RECORD NEW_RECORD_ALIAS NEW_RECORD</pre>
%mergerecord	<p>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 all 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).</p> <pre data-bbox="491 866 980 892">%mergerecord OLD_RECORD NEW_RECORD</pre> <p>If there are duplicate fields in the records being merged the original record's fields will not be overwritten.</p>
%copyrecord	<p>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.</p> <pre data-bbox="491 1130 966 1156">%copyrecord NEW_RECORD OLD_RECORD</pre>
%realformat	<p>Specifies how to format real variables. To format in exponential notation with 16 digits of precision, use</p> <pre data-bbox="483 1269 839 1295">%realformat "EXPONENTIAL"</pre> <p>To format without loss of precision and minimal number of characters, use</p> <pre data-bbox="483 1402 780 1428">%realformat "CONCISE"</pre> <p>When inlining S-functions, the format is set to concise. You can switch to exponential, but should switch it back to concise when done.</p>

Target Language Compiler Directives (Continued)

Directive	Description
%language	<p>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.</p>
%implements	<p>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, this should be the first noncomment line in the file as in</p> <pre data-bbox="480 652 955 682"> %implements "s_function_name" "C" </pre> <p>The next noncomment lines will be %function directives specifying the functionality of the S-function.</p> <p>See the %language and GENERATE function descriptions for further information.</p>
%generatefile	<p>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</p> <pre data-bbox="480 1065 970 1095"> %generatefile "Sin" "sin_wave.tlc" </pre>

Target Language Compiler Directives (Continued)

Directive	Description
%filescope	<p>Limits the scope of variables to the file in which they are defined. A %filescope directive, anywhere in a file declares that all variables in the file are visible only within that file. Note that this limitation also applies to any files inserted, via the %include directive, into the file containing the %filescope directive.</p> <p>The %filescope directive should not be used within functions or GENERATE functions.</p> <p>%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.</p>
%include %addincludepath	<p>%include "file.tlc" — insert specified target file at the current point. Use %addincludepath "directory" to add additional paths to be searched. We recommend UNIX-style forward slashes for directory names, as they will work on both UNIX and PC systems. However, if you do use backslashes in PC directory names, be sure to escape them, e.g., "C:\\mytlc". Alternatively, you can express a PC directory name as a literal using the L format specifier, as in L"C:\\mytlc". All %include directives behave as if they were in a global context, such that</p> <pre data-bbox="491 1130 836 1190">%addincludepath "./sub1" %addincludepath "./sub2"</pre> <p>in a .tlc file enables either subdirectory to be referenced implicitly:</p> <pre data-bbox="491 1272 880 1333">%include "file_in_sub1.tlc" %include "file_in_sub2.tlc"</pre>

Target Language Compiler Directives (Continued)

Directive	Description
<pre>%roll %endroll</pre>	<p data-bbox="454 343 1326 505">Multiple inclusion plus intrinsic loop rolling based upon a specified threshold. This directive can be used by most Simulink blocks which have the concept of an overall block width that is usually the width of the signal passing through the block. An example of the %roll directive is for a gain operation, $y=u*k$:</p> <pre data-bbox="488 534 1326 878"> %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 </pre> <p data-bbox="454 907 1326 1237">The %roll directive is similar to %foreach, except 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 we have 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.</p> <p data-bbox="454 1263 1326 1501">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 into a loop. When the decision is made to place a given roll region into a loop, the loop control variable will be a valid identifier (e.g., "i"), otherwise it will be "".</p>

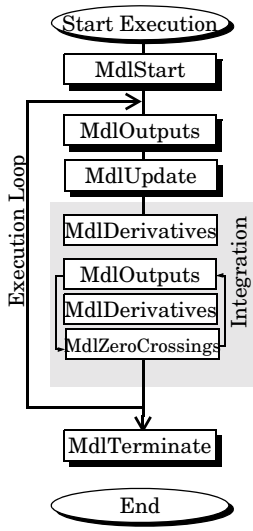
Target Language Compiler Directives (Continued)

Directive	Description
<p><code>%roll</code> (continued)</p>	<p>The block parameter is the current block that is being rolled. The "Roller" parameter specifies the name for internal GENERATE_TYPE calls made by <code>%roll</code>. The default <code>%roll</code> handler is "Roller", which is responsible for setting up the default block loop rolling structures (e.g., a C for loop).</p> <p>The <code>rollVars</code> (roll variables) are passed to "Roller" functions to create the correct roll structures. The defined loop variables relative to a block are</p> <ul style="list-style-type: none"> "U" All inputs to the block. It assumes you use <code>LibBlockInputSignal(portIdx, "", lcv, sigIdx)</code> to access each input, where <code>portIdx</code> starts at 0 for the first input port. "ui" Similar to "U", except only for specific input, <i>i</i>. The "u" must be lower case or it will be interpreted as "U" above. "Y" All outputs of the block. It assumes you use <code>LibBlockOutputSignal(portIdx, "", lcv, sigIdx)</code> to access each output, where <code>portIdx</code> starts at 0 for the first output port. "yi" Similar to "Y", except only for specific output, <i>i</i>. The "y" must be lower case or it will be interpreted as "Y" above. "P" All parameters of the block. It assumes you use <code>LibBlockParameter(name, "", lcv, sigIdx)</code> to access them. "<param>/name" Similar to "P", except specific for a specific name. rwork All RWork vectors of the block. It assumes you use <code>LibBlockRWork(name, "", lcv, sigIdx)</code> to access them. "<rwork>/name" Similar to RWork, except for a specific name. dwork All DWork vectors of the block. It assumes you use <code>LibBlockDWork(name, "", lcv, sigIdx)</code> to access them. "<dwork>/name" Similar to DWork, except for a specific name. iwork All IWork vectors of the block. It assumes you use <code>LibBlockIWork(name, "", lcv, sigIdx)</code> to access them. "<iwork>/name" Similar to IWork, except for a specific name. pwork All PWork vectors of the block. It assumes you use <code>LibBlockPWork(name, "", lcv, sigIdx)</code> to access them. "<pwork>/name" Similar to PWork, except for a specific name. "Mode" The mode vector. It assumes you use <code>LibBlockMode("", lcv, sigIdx)</code> to access it. "PZC" Previous zero crossing state. It assumes you use <code>LibPrevZCState("", lcv, sigIdx)</code> to access it.

Target Language Compiler Directives (Continued)

Directive	Description
%roll <i>(continued)</i>	<p>To <i>roll</i> 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,</p> <pre> datatype *buf = (datatype*)%<LibBlockPWork(name,"","",0) %roll sigIdx = RollRegions, lcv = RollThreshold, block, ... "Roller", rollVars *buf++ = whatever; %endroll </pre> <p>Note: In the above example, sigIdx and lcv are local to the body of the loop.</p>
%breakpoint	<p>Sets a breakpoint for the TLC debugger. See “%breakpoint Directive” on page 6-8.</p>
%function %return %endfunction	<p>A function that returns a value is defined as</p> <pre> %function name(optional-arguments) %return value %endfunction </pre> <p>A void function does not produce any output and is not required to return a value. It is defined as</p> <pre> %function name(optional-arguments) void %endfunction </pre> <p>A function that produces outputs to the current stream and is not required to return a value is defined as</p> <pre> %function name(optional-arguments) Output %endfunction </pre>

Target Language Compiler Directives (Continued)

Directive	Description
 <pre> graph TD Start([Start Execution]) --> MdlStart[MdlStart] MdlStart --> MdlOutputs1[MdlOutputs] MdlOutputs1 --> MdlUpdate[MdlUpdate] MdlUpdate --> MdlDerivatives1[MdlDerivatives] MdlDerivatives1 --> MdlOutputs2[MdlOutputs] MdlOutputs2 --> MdlDerivatives2[MdlDerivatives] MdlDerivatives2 --> MdlZeroCrossings[MdlZeroCrossings] MdlZeroCrossings --> MdlDerivatives1 MdlDerivatives1 --> MdlTerminate[MdlTerminate] MdlTerminate --> End([End]) </pre>	<p>For block target files, you can add to your inlined .t1c file the following functions that will get called by the model wide target files during code generation</p> <pre> %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 enables. 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. </pre>
<pre> %function %return %endfunction (continued) </pre>	

Target Language Compiler Directives (Continued)

Directive	Description
	<p><code>%function InitializeConditions(block, system) Output</code> 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 <code>MdlStart</code> and/or subsystem initialization routines.</p> <p><code>%function Outputs(block, system) Output</code> The primary function of your block. Place in <code>MdlOutputs</code>.</p> <p><code>%function Update(block, system) Output</code> Use this function if your block has actions to be performed once per simulation loop, such as updating discrete states. Place in <code>MdlUpdate</code></p> <p><code>%function Derivatives(block, system) Output</code> Used this function if your block has derivatives for <code>MdlDerivatives</code>.</p> <p><code>%function ZeroCrossings(block, system) Output</code> Used this function if your block does zero crossing detection and has actions to be performed in <code>MdlZeroCrossings</code>.</p> <p><code>%function Terminate(block, system) Output</code> Use this function if your block has actions that need to be in <code>MdlTerminate</code>.</p>
<p><code>%foreach</code> <code>%endforeach</code></p>	<p>Multiple inclusion that iterates from 0 to the <code>upperLimit-1</code> constant integer expression. Each time through the loop, the <code>loopIdentifier</code>, (e.g., <code>x</code>) is assigned the current iteration value.</p> <pre> %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 </pre> <p>Note: The <code>upperLimit</code> expression is cast to a TLC integer value. The <code>loopIdentifier</code> is local to the loop body.</p>

Target Language Compiler Directives (Continued)

Directive	Description
<p><code>%for</code></p>	<p>Multiple inclusion directive with syntax</p> <pre> %for ident1 = const-exp1, const-exp2, ident2 = const-exp3 %body %break %continue %endbody %endfor </pre> <p>The first portion of the <code>%for</code> directive is identical to the <code>%foreach</code> statement. The <code>%break</code> and <code>%continue</code> directives act the same as they do in the <code>%foreach</code> directive. <code>const-exp2</code> is a Boolean expression that indicates whether the loop should be rolled (see <code>%roll</code> above).</p> <p>If <code>const-exp2</code> evaluates to <code>TRUE</code>, <code>ident2</code> is assigned the value of <code>const-exp3</code>. Otherwise, <code>ident2</code> is assigned an empty string.</p> <p>Note: <code>ident1</code> and <code>ident2</code> above are local to the loop body.</p>
<p><code>%openfile</code> <code>%selectfile</code> <code>%closefile</code></p>	<p>These are used to manage the files that are created. The syntax is</p> <pre> %openfile streamId="filename.ext" mode {open for writing} %selectfile streamId {select an open file} %closefile streamId {close an open file} </pre> <p>Note that the “<code>filename.ext</code>” is optional. If no filename is specified, a variable (string buffer) named <code>streamId</code> is created containing the output. The mode argument is optional. If specified, it can be “<code>a</code>” for appending, “<code>r</code>” for reading, or “<code>w</code>” for writing.</p> <p>Note that the special <code>streamId</code> <code>NULL_FILE</code> specifies that no output occur. The special <code>streamId</code> <code>STDOUT</code> specifies output to the terminal.</p>

Target Language Compiler Directives (Continued)

Directive	Description
	<p>To create a buffer of text, use</p> <pre>%openfile buffer text to be placed in the 'buffer' variable. %closefile buffer</pre> <p>Now buffer contains the expanded text specified between the %openfile and %closefile directives.</p>
%generate	<pre>%generate blk fn</pre> <p>is equivalent to <code>GENERATE(blk,fn)</code>.</p> <pre>%generate blk fn type</pre> <p>is equivalent to <code>GENERATE(blk,fn,type)</code>.</p> <p>See “GENERATE and GENERATE_TYPE Functions” on page 5-35.</p>
%undef	<pre>%undef var</pre> <p>removes the variable <code>var</code> from scope. If <code>var</code> is a field in a record, %undef removes that field from the record. If <code>var</code> is a record array, %undef removes the first element of the array.</p>

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 to delimit comments within your code. Use the `%%` construct for line-based comments; all 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 5-66.

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 up the directive on to multiple lines. For example:

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

or

```
%roll sigIdx = RollRegions, lcv = RollThreshold, block,...\n    "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 Values

This table shows the types of values you can use within the context of expressions in your target language files. All expressions in the Target Language Compiler must use these types.

Target Language Values

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	A 64-bit double-precision complex number (double on the target machine)
"Complex32"	3.0F+5.0Fi	A 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...	A user-defined function and TLC_FALSE otherwise
"Gaussian"	3+5i	A 32-bit integer imaginary number (int on the target machine)
"Identifier"	abc	Identifier values can only appear 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 as appropriate 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 type except vectors or matrices. The Matrix (3,2) text in the example is optional.
"Number"	15	An integer number (int on the target machine)

Target Language Values (Continued)

Value Type String	Example	Description
"Range"	[1:5]	A range of integers between 1 and 5, inclusive
"Real"	3.14159	A floating-point number (double on the target machine), including exponential notation
"Real32"	3.14159F	A 32-bit single-precision floating-point number (float on the target machine)
"Scope"	Block { ... }	A block record
"Special"	FILE_EXISTS	A 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 all of the ANSI C standard escape sequences such as \n, \r, \t, etc. Use of line continuation characters (i.e. \ and ...) inside of strings is illegal.
"Subsystem"	<sub1>	A subsystem identifier. Within the context of an expansion, be careful to escape the delimiters on a subsystem identifier as in: %<x == <sub\>>.
"Unsigned"	15U	A 32-bit unsigned integer (unsigned int on the target machine)
"Unsigned Gaussian"	3U+5Ui	A 32-bit complex unsigned integer (unsigned int on the target machine)
"Vector"	[1, 2] or Vector(2) [1, 2]	Vectors are lists of values. The individual elements of a vector do not need to be the same type, and may be any type except vectors or matrices.

Target Language Expressions

In any place throughout a target file, you can include an expression of the form `%<expression>`. 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 any line to perform textual substitution. To include the `>` character within a replacement, you must escape it with a “\” character as in

```
%<x \> 1 ? "ABC" : "123">
```

The Target Language Expressions table 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`) will not be 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 5-26 for information on the promotions that result when the Target Language Compiler operates on mixed types of expressions.

Target Language Expressions

Expression	Definition
constant	Any constant parameter value, including vectors and matrices
variable-name	Any valid in-scope variable name, including the local function scope, if any, and the global scope
::variable-name	Used within a function to indicate that the function scope is ignored when looking up the variable. 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 of the parentheses is the function/macro name; the expressions inside are the arguments to the function or macro. Note: Since macros are text-based, they cannot be used within the same expression as other operators.

Target Language Expressions (Continued)

Expression	Definition
<code>expr . expr</code>	The first expression must have a valid scope; the second expression is a parameter name within that scope.
<code>(expr)</code>	Use <code>()</code> to override the precedence of operations.
<code>!expr</code>	Logical negation (always generates <code>TLC_TRUE</code> or <code>TLC_FALSE</code>). The argument must be numeric or Boolean.
<code>-expr</code>	Unary minus negates the expression. The argument must be numeric.
<code>+expr</code>	No effect; the operand must be numeric.
<code>~expr</code>	Bitwise negation of the operand. The argument must be integral.
<code>expr * expr</code>	Multiply the two expressions together; the operands must be numeric.
<code>expr / expr</code>	Divide the two expressions; the operands must be numeric.
<code>expr % expr</code>	Take the integer modulo of the expressions; the operands must be integral.

Target Language Expressions (Continued)

Expression	Definition
expr + expr	<p>Works on numeric types, strings, vectors, matrices, and records as follows:</p> <p>Numeric Types - Add the two expressions together; the operands must be numeric.</p> <p>Strings - The strings are concatenated.</p> <p>Vectors - If the first argument is a vector and the second is a scalar, it appends the scalar to the vector.</p> <p>Matrices - If the first argument is a matrix and the second is a vector of the same column-width as the matrix, it appends the vector as another row in the matrix.</p> <p>Records - If the first argument is a record, it adds the second argument as a parameter identifier (with its current value).</p> <p>Note, the addition operator is associative.</p>
expr - expr	<p>Subtracts the two expressions; the operands must be numeric.</p>
expr << expr	<p>Left shifts the left operand by an amount equal to the right operand; the arguments must be integral.</p>
expr >> expr	<p>Right shifts the left operand by an amount equal to the right operand; the arguments must be integral.</p>
expr > expr	<p>Tests if the first expression is greater than the second expression; the arguments must be numeric.</p>

Target Language Expressions (Continued)

Expression	Definition
<code>expr < expr</code>	Tests if the first expression is less than the second expression; the arguments must be numeric.
<code>expr >= expr</code>	Tests if the first expression is greater than or equal to the second expression; the arguments must be numeric.
<code>expr <= expr</code>	Tests if the first expression is less than or equal to the second expression; the arguments must be numeric.
<code>expr == expr</code>	Tests if the two expressions are equal.
<code>expr != expr</code>	Tests if the two expression are not equal.
<code>expr & expr</code>	Performs the bitwise AND of the two arguments; the arguments must be integral.
<code>expr ^ expr</code>	Performs the bitwise XOR of the two arguments; the arguments must be integral.
<code>expr expr</code>	Performs the bitwise OR of the two arguments; the arguments must be integral.
<code>expr && expr</code>	Performs the logical AND of the two arguments and returns <code>TLC_TRUE</code> or <code>TLC_FALSE</code> . This can be used on either numeric or Boolean arguments.
<code>expr expr</code>	Performs the logical OR of the two arguments and returns <code>TLC_TRUE</code> or <code>TLC_FALSE</code> . This can be used on either numeric or Boolean arguments.

Target Language Expressions (Continued)

Expression	Definition
<code>expr ? expr : expr</code>	Tests the first expression for TLC_TRUE. If true, the first expression is returned; otherwise the second expression is returned.
<code>expr , expr</code>	Returns the value of the second expression.

Note Relational operators (`<`, `=<`, `>`, `>=`, `!=`, `==`) can be used with nonfinite values.

Reminder It is not necessary 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 result 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 expression.

For example, if the operation involves a Boolean expression (B) and an unsigned expression (U), the result will be an unsigned expression (U).

Table 5-1: Datatypes 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
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 all 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 a set of 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
```

and

```
%switch constant-expression  
%case constant-expression  
%break  
%default  
%endswitch
```

%if

The `constant-expression` must evaluate to an integral expression. It controls the inclusion of all the following lines until it encounters a `%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 any other integral value, the lines following the `%if` directive are included up until the `%endif`, `%elseif`, or `%else` directives.

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 up until the subsequent `%else`, `%elseif`, or `%endif` directive.

The `%else` directive begins the inclusion of source text if all of 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 5-21.

%switch

The `%switch` statement evaluates the constant expression and compares it to all 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` will exit 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 integral 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:

- `%break` can be used to exit the nearest enclosing `%for`, `%foreach`, or `%switch` statement.
- `%continue` can be used to begin the next iteration of a loop.

%for

Note The `%for` directive is functional, but it is not recommended. Rather, use `%roll`, which provides the same capability in a more open way. The Real-Time Workshop does not make use of 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 the `%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, all of 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 all of 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-exp` and 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 in all respects 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 since 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. Any extra arguments passed on 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 any language-specific declarations, loop code, and so on as required to generate correct 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 Real-Time Workshop has the capability to extract references to the block I/O and other Real-Time Workshop vectors that vastly simplify the body of the `%roll` statement. These functions include `LibBlockInputSignal`, `LibBlockOutputSignal`, `LibBlockParameter`, `LibBlockRWork`, `LibBlockIWork`, `LibBlockPWork`, and `LibDeclareRollVars`, `LibBlockMatrixParameter`, `LibBlockParameterAddr`, `LibBlockContinuousState`, and `LibBlockDiscreteState` function reference pages in “TLC Function Library Reference” on page 8-1. 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 ensure that the correct implementation files are 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"
```

All blocks in Simulink have a Type parameter. This parameter is a string that specifies the type of the block, e.g., "Sin" or "Gain". The object-oriented facility uses this type to search the path for a file that implements the correct 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 a %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 any 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 5-56.) 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 Real-Time Workshop. 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. Similar to 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 if 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. All 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, and if this file is the currently selected stream, `%closefile` invokes `%selectfile` to reselect the last 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 all 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:\\directory1\\directory2"      (PC)
%addincludepath "/directory1/directory2"         (UNIX)
```

To specify a relative path, the path must explicitly start with `“.”`. For example:

```
%addincludepath ".\\directory2"                 (PC)
%addincludepath "./directory2"                 (UNIX)
```

Note that for PC, the backslashes must be escaped (doubled).

When an explicit relative path is specified, the directory 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 directories in the following order for target or include files:

- 1** The current directory
- 2** Any `%addincludepath` directives
- 3** Any include paths specified at the command line via `-I` (in reverse order)

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.

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. All of 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 5-71 for additional information about switches.

The %assert directive will evaluate the expression and will produce a stack trace if the expression evaluates to a Boolean false.

Note In order for %assert directives to be evaluated, **Enable TLC Assertions** must be selected on the **TLC debugging** section of the Real-Time Workshop pane. The default action is for asserts 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
<code>CAST(expr, expr)</code>	<p>The first expression must be a string that corresponds to one of the type names in the Target Language Values table, 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</p> <pre>CAST("Real", variable-name)</pre> <p>An example of this is in working with parameter values for S-functions. To use them within C code, you need to typecast them to real so that a value such as “1” will be formatted as “1.0” (see also <code>%realformat</code>).</p>
<code>EXISTS(var)</code>	<p>If the <code>var</code> identifier is not currently in scope, the result is <code>TLC_FALSE</code>. If the identifier is in scope, the result is <code>TLC_TRUE</code>. <code>var</code> can be a single identifier or an expression involving the <code>.</code> and <code>[]</code> operators.</p> <p>Note: prior to TLC release 4, the semantics of <code>EXISTS</code> differed from the above. See “Compatibility Issues” on page 1-19.</p>
<code>FEVAL(matlab-command, TLC-expressions)</code>	<p>Performs an evaluation in MATLAB. For example</p> <pre>%assign result = FEVAL("sin",3.14159)</pre> <p>The <code>%matlab</code> directive can be used to call a MATLAB function that does not return a result. For example:</p> <pre>%matlab disp(2.718)</pre> <p>Note: if the MATLAB function returns more than one value, TLC only receives the first value.</p>
<code>FILE_EXISTS(expr)</code>	<p><code>expr</code> must be a string. If a file by the name <code>expr</code> does not exist on the path, the result is <code>TLC_FALSE</code>. If a file by that name exists on the path, the result is <code>TLC_TRUE</code>.</p>

TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
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 a array of strings containing the record field names associated with the record. As it returns a sorted list of strings, 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. By virtue of hash lookup, executes in constant time
GENERATE(record, function-name, ...)	This is used to execute function calls mapped to a specific record type (i.e., Block record functions). For example, use this to execute the functions in the .t1c files for built-in blocks. Note, 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 .t1c 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, which 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.

TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
GENERATE_FUNCTION_EXISTS (record, function-name)	Determines if a given block function exists. The first expression is the same as the first argument to GENERATE, 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 type overrides the Type field of the record. Use this when executing functions mapped to specific S-function blocks records based upon the S-function name (i.e., name becomes type).
GENERATE_TYPE_FUNCTION_EXISTS S (record, function-name, type)	Same as GENERATE_FUNCTION_EXISTS except it overrides the Type built into the record.
GET_COMMAND_SWITCH	Returns the value of command-line switches. Only the following switches are supported: v, m, p, 0, d, dr, r, I, a See also “Command Line Arguments” on page 5-71.
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

TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
INT32MIN	-2147483648
INTMIN	Minimum integer value on target machine
INTMAX	Maximum integer value on target 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 datatypes of both expressions are numeric: returns TLC_TRUE if the first expression contains the same value as the second expression; returns TLC_FALSE otherwise. Where the datatype of either expression is non-numeric (e.g. string or record): returns TLC_TRUE if and only if both expressions have the same datatype 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 to 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.

TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
ISSLPRMREF(param.value)	<p>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. Example:</p> <pre>%if !ISSLPRMREF(param.Value) assign param.Value = CAST("Real", param.Value) %endif</pre>
NULL_FILE	<p>A predefined file for no output that you can use as an argument to %selectfile to prevent output.</p>
NUMTLCFILES	<p>The number of target files used thus far in expansion.</p>
OUTPUT_LINES	<p>Returns the number of lines that have been written to the currently selected file or buffer. Does not work for STDOUT or NULL_FILE</p>
REAL(expr)	<p>Returns the real part of a complex number.</p>
REMOVEFIELD(record, "fieldname")	<p>Removes the specified field from the contents of the record. Returns TLC_TRUE if the field was removed; otherwise returns TLC_FALSE.</p>
ROLL_ITERATIONS()	<p>Returns the number of times the current roll regions are looping or NULL if not inside a %roll construct.</p>
SETFIELD(record, "fieldname", value)	<p>Sets the contents of the field name associated to the record. Returns TLC_TRUE if the field was added; otherwise returns TLC_FALSE.</p>

TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
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 integral index into this row vector; otherwise the entire row vector is returned. SIZE(x) applied to any scalar returns [1 1]. SIZE(x) applied to any scope returns the number of repeated entries of that scope type (e.g., 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 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). All the ANSI escape sequences are translated into string form.
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 in Real-Time Workshop.

TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
SYSNAME (expr)	<p>Looks for specially formatted strings of the form <x>/y and returns x and y as a 2-element string vector. This is used to resolve subsystem names in Real-Time Workshop. For example,</p> <pre>%<sysname ("<sub>/Gain")></pre> <p>returns</p> <pre>["sub", "Gain"]</pre> <p>In Block records, the name of the block is written similar to <sys/blockname> where sys is S# or Root. You can obtain the full pathname by calling LibGetBlockPath(block); this will include newlines and other troublesome characters that cause display difficulties. To obtain a full pathname suitable for one line comments but not identical to the Simulink pathname, use LibGetFormattedBlockPath(block).</p>
TLCFILES	Returns a vector containing the names of all the target files included thus far in the expansion. Absolute paths are used. See also NUMTLCFILES.
TLC_FALSE	Boolean constant which equals a negative evaluated Boolean expression.
TLC_TRUE	Boolean constant which equals a positive evaluated Boolean expression.
TLC_TIME	The date and time of compilation.
TLC_VERSION	The 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 value type string in the table “Target Language Values,” for possible values.

TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
UINT8MAX	255U
UINT16MAX	65535U
UINT32MAX	4294967295U
UINTMAX	Maximum unsigned integer value on target machine.
WHITE_SPACE(expr)	Accepts a string and returns 1 if the string contains only whitespace 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 M-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 calling MATLAB.

This table shows the conversions that are made when calling MATLAB.

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)

MATLAB Conversions (Continued)

TLC Type	MATLAB Type
"Unsigned"	Double (scalar or Matrix)
"String"	String
"Vector"	If the vector is homogeneous, it will convert to a MATLAB vector of the appropriate value. If the vector is heterogeneous, it converts 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
"Matrix"	If the matrix is homogeneous, it will convert to a MATLAB matrix of the appropriate value. If the matrix is heterogeneous, it converts 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
Any other type	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)
Complex double precision	Complex (scalar or Matrix)
Sparse matrix	Expanded out 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

More Conversions (Continued)

MATLAB Type	TLC Type
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 one.

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 Constant(s)	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 version 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 (if any), file function scope (if any), generate file scope (if any), 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. Existing identifiers can be changed by completely respecifying them. The constant expressions can include any 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 always 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 always 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.

In order 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 Real-Time Workshop 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

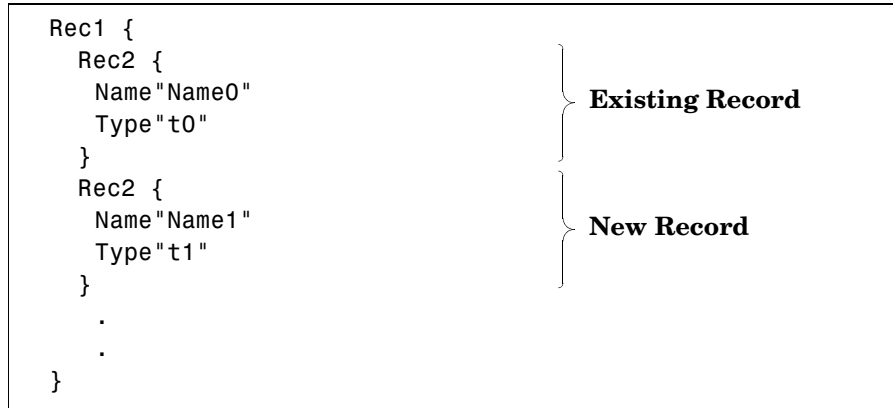
```
%createrecord Rec { Name "Name"; Type "t" }
```

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

```

Rec1 {
  Rec2 {
    Name "Name0"
    Type "t0"
  }
  Rec2 {
    Name "Name1"
    Type "t1"
  }
  Rec2 {
    Name "Name2"
    Type "t2"
  }
  .
  .
}

```

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.

```

Block {
  .
  .
  .
  N 500
}

```

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” below.

The term scope also refers to a collection, or *pool*, of variables that have the same scope. At any point in the execution of a TLC program, several scopes may 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) would also exist.

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 5-61.

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 in particular that the `CompiledModel` record of the `model.rtw` file has global scope. Therefore, you can access this structure from any of 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 5-61 for examples.

Note that you can use the `%undef` directive to free up 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 up 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 5-61 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 5-58.

%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 5-35). This section will discuss 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 resolving variable references. Thus the record (blk) is visible to the dispatched function, as if there were an implicit `%with <blk>... %endwith` directive in the dispatched function.

In this context, the record (blk) is said to be in *generate scope*.

Three TLC files, demonstrating 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 the appropriate 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 is `polymorph.tlc`.

```
%% polymorph.tlc

%language "C"

%%create records used as scopes within the 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 is `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 is 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 on the MATLAB console, 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 looking 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 sideeffect(arg)
    %assign ::theglobal = arg    %% this creates a global variable
%endfunction
```

How TLC Resolves Variable References

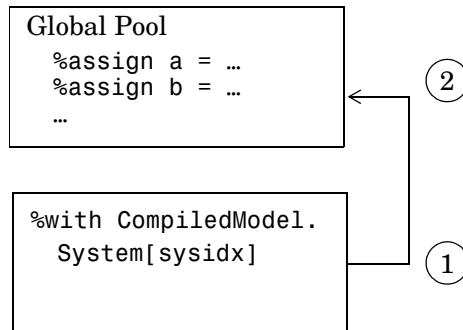
This section discusses how the Target Language Compiler searches the existing scopes to resolve variable references.

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.



%with Scope Added to Search Sequence

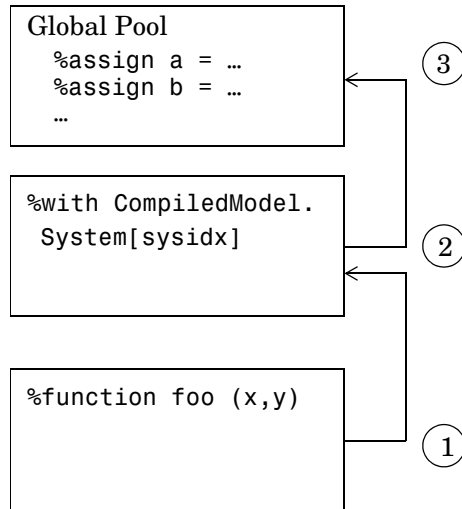
This technique makes it simpler to access embedded definitions. Using the `%with` construct (as in the previous example), you can refer to the system name simply by

```
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.



Scoping Rules Within Functions

For example, in the following code fragment:

```

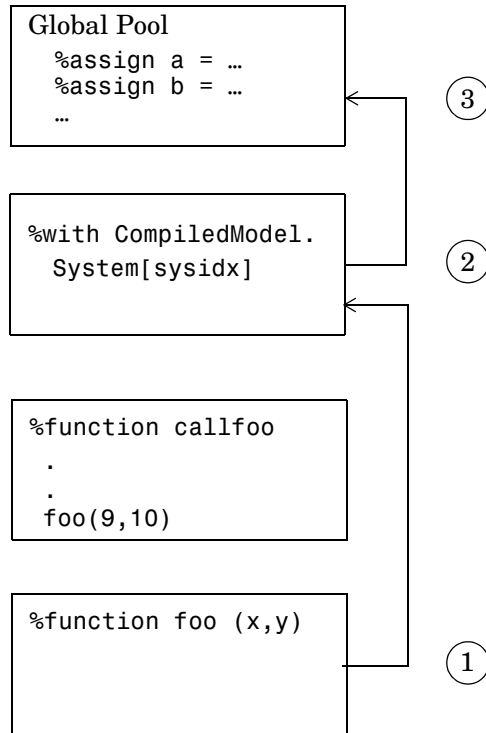
% with CompiledModel.System[sysidx]
.
.
.
    %assign a=foo(x,y)
.
.
.
%endwith
.
.
.
%function foo (a,b)
.
.
.

```

```
        assign myvar=Name
    .
    .
    .
    %endfunction
    .
    .
    .
    %<foo(1,2)>
```

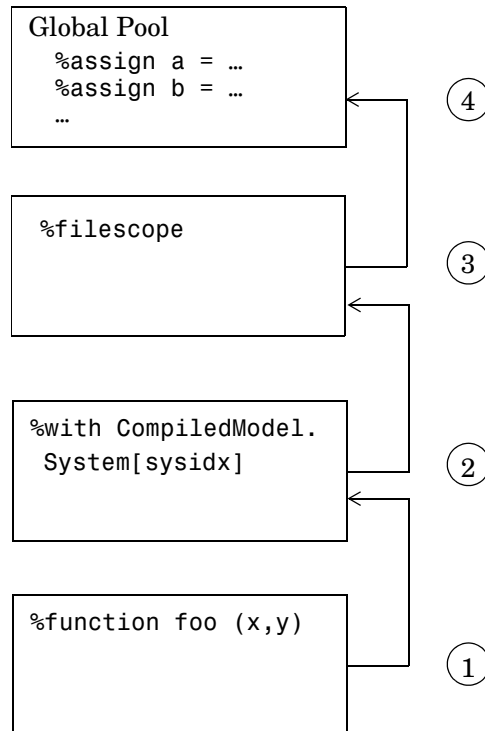
If `Name` is not defined in `foo`, the assignment will use the value of `Name` from the previous scope, `CompiledModel.System[SysIdx].Name`.

In the case of nested functions, only the innermost nested function scope is searched. In the diagram below, `foo` is called by `callfoo`. When resolving variable references in `foo`, only the scope of `foo` is searched (together with enclosing `%with` and global scopes.)



Nested File Scopes

File Scope. File scopes are searched before the global scope, as shown in the following diagram.



File Scopes Searched Before Global Scope

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 any output, unless they explicitly use the `%openfile`, `%selectfile`, and `%closefile` directives, or are output functions.

A function optionally returns a value with the `%return` directive. The returned value can be any of the types defined in the table “Target Language Values”.

In this example, a function, `name`, returns `x`, if `x` and `y` are equal, and 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 any context where variables are allowed.

All `%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 any `%with` statements appearing within the function.

Assignments to variables within a function always create new, local variables and can not change the value of global variables unless you use the scope resolution operator (`::`).

By default, a function returns a value and does not produce any 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, if any, 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 any output, as in

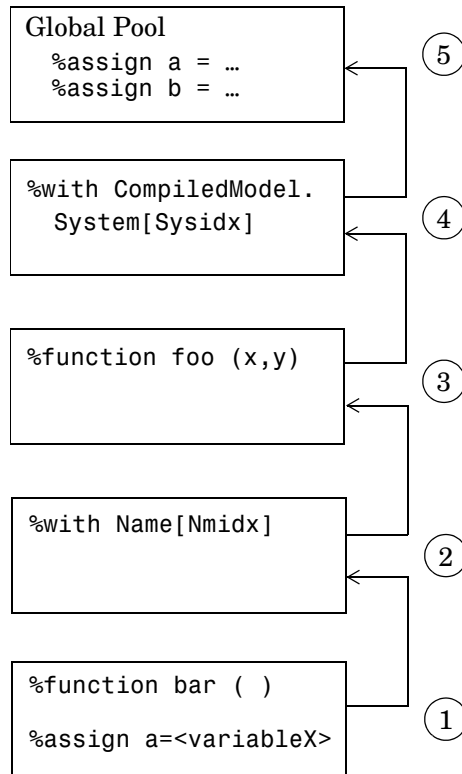
```
%function foo() void
...
%endfunction
```

Variable Scoping Within Functions

Within a function, the left-hand member of any `%assign` statement defaults to create a local variable. A new entry is created in the function's block within the scope chain; it does not affect any of the other entries. An example is shown in Figure , Scoping Rules Within Functions, on page 5-63.

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. Also, if a `%with` is included within a function, its associated scope is carried with any nested function call, as shown in Figure , Scoping Rules When Using `%with` Within a Function, on page 5-69.



Scoping Rules When Using %with Within a Function

%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 with no return value.

Command Line Arguments

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 makes no 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 a .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.
-Ipath	Adds the specified directory to the list of paths to be searched for TLC files.
-Opath	Specifies that all output produced should be placed in the designated directory, including files opened with %openfile and %closefile, and .log files created in debug mode. To place files in the current directory, use -O (use the capital letter O, not zero).
-m[number]	Specifies the maximum number of errors to report is <number>. If no -m argument appears on the command line, it defaults to reporting the first five errors. If the <number> argument is omitted on this option, 1 is assumed.
-x0	Parse TLC file only (do not execute).
-lint	Performs some simple checks for performance and deprecated features.

Target Language Compiler Switches (Continued)

Switch	Meaning
-p[number]	Print a dot ('.') indicating progress for every <number> of TLC primitive operations executed.
-d[a c f n o]	Invoke the TLC's debug mode. -da makes TLC execute any %assert directives. However, when building from within RTW, this flag is not needed and will be ignored, as it is superseded by the Enable TLC Assertions check box on the TLC debugging section of the Real-Time Workshop pane. -dc invokes the TLC command line debugger. -df filename invokes the TLC debugger and runs a debugger script as specified by filename. A debugger script is a text file containing valid debugger commands. TLC searches only the current working directory for the script file. -dn will cause TLC to produce log files indicating which lines were and were not hit during compilation. -do will disable the TLC debugging behavior.
-dr	Check 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.

As an example, the command line

```
tlc -r Demo.rtw -v grt.tlc
```

specifies that Demo.rtw should be read and used to process grt.tlc in verbose mode.

Filenames and Search Paths

All target files have the `.t1c` 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 all target files along this path. If you specify additional search paths with the `-I` switch of the `t1c` command or via the `%addincludepath` directive, they will be searched after the current working directory, and in the order in which you specify them.

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 (p. 6-2)

Using the TLC Debugger (p. 6-3)

TLC Coverage (p. 6-9)

TLC Profiler (p. 6-13)

Introducing the TLC debugging facility

Enabling tracing and coverage, and command summary

Determining what TLC statements are executed

Measuring the execution time of each TLC function

About the TLC Debugger

The TLC debugger helps you identify programming errors in your TLC code. Using the debugger, you can

- View the TLC call stack.
- Execute TLC code line-by-line and analyze and/or change variables in a specified block scope.

The TLC debugger has a command line interface and uses 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 will be silent otherwise. You may use `%warning` instead of `%trace` to print variables, but you will need to remove or comment out such lines after you are through debugging.

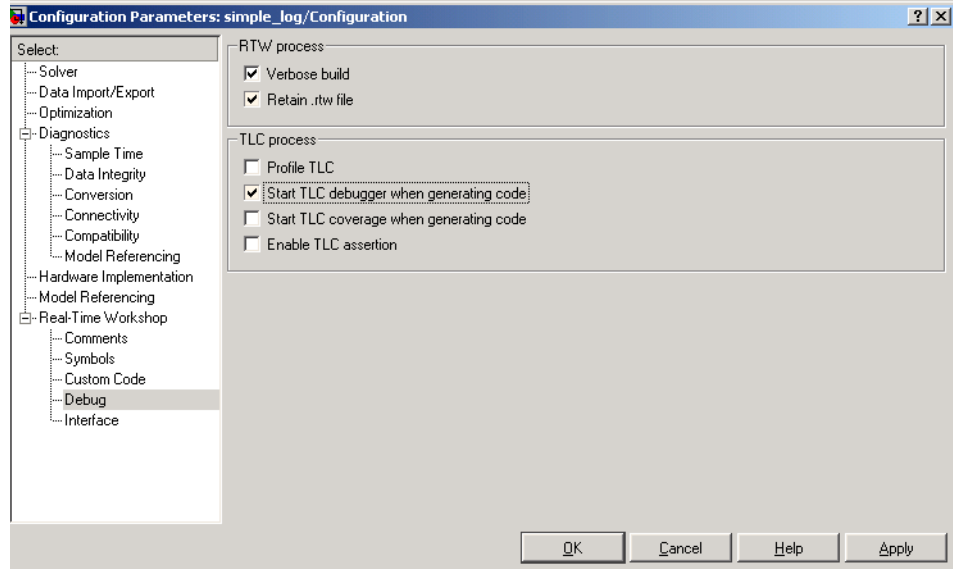
- 3** Use the TLC coverage log files to ensure that most parts of your code have been exercised.

Using the TLC Debugger

This section describes the basic procedures and commands for using the TLC debugger to identify bugs and potential problems in your TLC files. If you are simply modifying TLC files and not changing your model, be sure to read the section “Avoiding Rebuilding Models When Debugging TLC” on page 6-4.

Invoking the Debugger

- 1 To configure TLC for debugging via the **Configuration Parameters** dialog, select **Debug** under **Real-Time Workshop**.
- 2 Select **Retain .rtw file** in the RTW process subpane. This ensures that the `model.rtw` file is not deleted after code generation.
- 3 Select **Start TLC debugger when generating code** in the TLC process subpane to invoke the TLC debugger when starting the code generation process. The dialog box looks like this.



Selecting **Start TLC debugger when generating code** is equivalent to adding `-dc` to the **RTW System target file** field in the Real-Time Workshop 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 Real-Time Workshop files, and explore variables in your TLC file using `print`, `which`, or `whos`.

An alternate way to invoke the TLC debugger is from the MATLAB prompt. (This assumes you retained the `model.rtw` file in the project directory.) To avoid making mistakes, we recommend copying the `tlc` command output by Real-Time Workshop 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 found in the table “Target Language Compiler Switches” in Chapter 5.

Avoiding Rebuilding Models When Debugging TLC

If you are debugging TLC scripts for code generation, you can speed up the edit-generate-inspect cycle when you generate code for models that are not changing between iterations. You can bypass rebuilding the model (the RTW process) if all you are doing is editing TLC files used to generate code.

To use this feature, select the **Retain .rtw** file option in the **Real-time Workshop/Debug** pane. The next time you build, the `model.rtw` file will be saved in your build directory, along with two other files:

- `runtlccmd.m`
- `tlccmd.mat`

From that point on, you can invoke the Target Language Compiler outside the build process and with the proper parameters by executing `runtlccmd.m`. The MAT-file is used to store the parameters used by the M-file in issuing the TLC command. You can rebuild the model as required, and this time-saving option will remain available as long as you continue to retain your `model.rtw` file each time you build.

TLC Debugger Command Summary

The table “TLC Debugger Commands” on page 6-6 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 any 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 trace function	Set a breakpoint. See also “%breakpoint Directive” on page 6-8
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.

TLC Debugger Commands

Command	Description
print expression	Print the value of a TLC expression. To print a record, you must specify a fully qualified “scope” such as <code>CompiledModel.System[0].Block[0]</code> .
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).
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 any point 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 in foo.tlc on error, use

```
TLC_DEBUG> break "foo.tlc":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
```

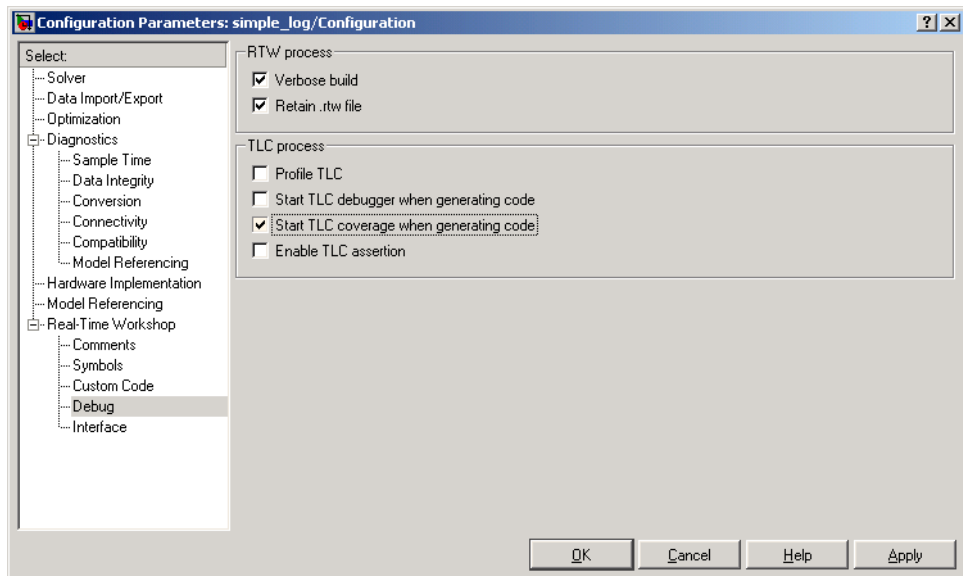
In this example, clear 2 clears the second breakpoint.

TLC Coverage

The example in the last section used the debugger to detect a problem in one section of the TLC file. Since it is conceivable that a test model does not cover all possible cases, there is a technique that traces the untested cases, the TLC coverage option.

Using 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 initiate TLC coverage generation, select **Start TLC coverage when generating code** from the TLC process subpane of the **Real-Time Workshop/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 the Real-Time Workshop created project directory 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 the `sfcdemo_sdotproduct` demo model, located in `matlabroot/toolbox/simulink/simdemos`. This model inlines the `sdotproduct` S-function in TLC. The TLC file that implements the S-function is located in `matlabroot/toolbox/simulink/blocks/tlc_c/`. The .log file for `sdotproduct.tlc` is `sdotproduct.log`, which is placed in your build directory. The contents of `sdotproduct.log` are:

```
Source: \\bat07\anightly\matlab\toolbox\simulink\blocks\tlc_c\sdotproduct.tlc
0: %% $RCSfile: sdotproduct.ttlc,v $
0: %% File : sdotproduct.tlc generated from sdotproduct.ttlc revision 1.6
0: %% $Date: 2002/04/10 18:17:59 $
0: %%
0: %% Murali Yeddanapudi, 27-May-1998
0: %% Copyright 1990-2002 The MathWorks, Inc.
0: %%
0: %% Abstract:
0: %%     Dot product block target file.
1:
1:
1: %implements sdotproduct "C"
1:
1:
0: %% Function: FcnThriftdComplexMultiply
=====
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 FcnThriftdComplexConjMultiply(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 over-written and u0im is non-zero
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 non-scalar 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:          %<FcnThriftdComplexConjMultiply(u0re, u0im, 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.

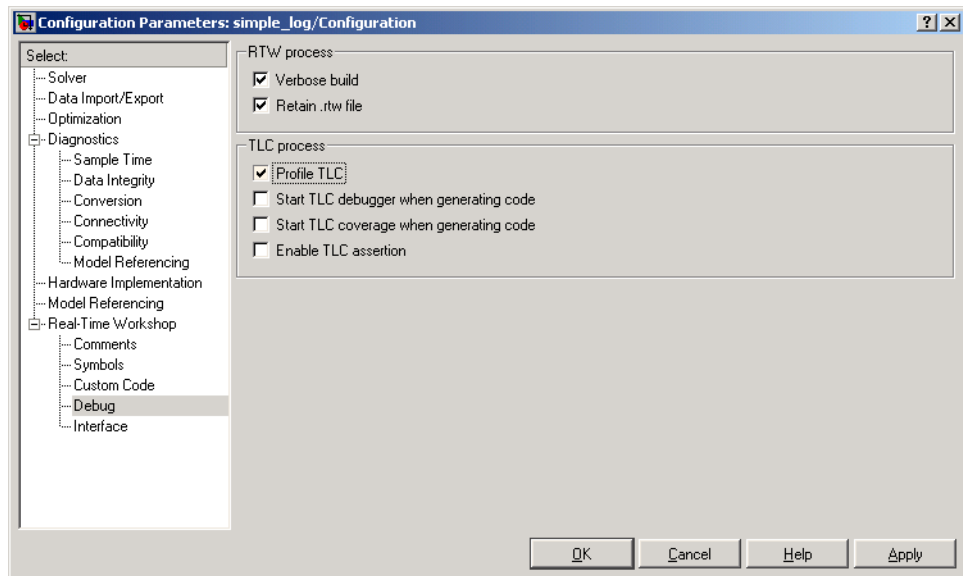
Looking at the `sdotproduct.log` file, you can see that the code has not been used to assign default values to parameters (e.g., the beginning part of the code for function `FcnThriftdComplexConjMultiply`). Using this log as a reference and creating models that exercise unexecuted lines, you can make sure that your code is more robust.

TLC Profiler

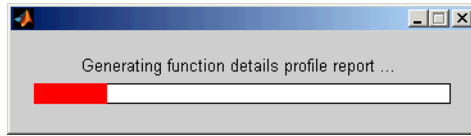
The TLC profiler collects timing statistics for TLC code. It collects execution time for functions, scripts, macros, and built-in functions. These results become the basis of HTML reports that are identical in format to MATLAB profiler reports. By analyzing the report, you can identify bottlenecks in your code that make code generation take longer.

Using the Profiler

To access the profiler, select **Profile TLC** from the TLC debugging category of the Real-Time Workshop 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 HTML summary and related files are created. A progress bar paces the report generation:



The profile report is generated into the Real-Time Workshop build directory. To open the report, cd to the build directory and open the file `model.html`, opening it in a browser window. Here is sample of a TLC profiling report:

[Summary](#) | [Function Details](#)

TLC Profile Report: Summary

Report generated 13-Apr-2004 12:07:13

Total recorded time: 6.57 s
 Number of Builtins: 22
 Number of Evals: 1
 Number of Generate Scripts: 7
 Number of Normal Functions: 74
 Number of Output Functions: 57
 Number of Scripts: 115
 Number of Void Functions: 498
 Clock precision: 0.0000001 s
 Clock Speed: 1000 Mhz

Function List

Name	Time	Time %	Calls	Time/call	Self time	Self %	Loc
codegenentry.tlc	6.5694464	100.0%	1	6.56944640000	0.0200288	0.3%	//
grt.tlc	6.5694464	100.0%	1	6.56944640000	0.0000000	0.0%	//
commonsetup.tlc	4.0858752	62.2%	1	4.08587520000	0.1802592	2.7%	//
funclib.tlc	3.3548240	51.1%	1	3.35482400000	2.1330672	32.5%	//
commonentry.tlc	2.4635424	37.5%	1	2.46354240000	0.0400576	0.6%	//
formatwide.tlc	2.4234848	36.9%	1	2.42348480000	0.1301872	2.0%	//
fixptlib.tlc	0.6309072	9.6%	1	0.63090720000	0.5708208	8.7%	//
SLibERTWriteSource	0.5507920	8.4%	1	0.55079200000	0.1602304	2.4%	//
SLibFormatBody	0.5307632	8.1%	1	0.53076320000	0.0000000	0.0%	//
%	0.3605184	5.5%	2077	0.00017357650	0.2904176	4.4%	Eva
SLibErtAutoFunctions	0.3004320	4.6%	1	0.30043200000	0.0000000	0.0%	//
SLibFormatExport	0.2904176	4.4%	1	0.29041760000	0.0000000	0.0%	//
SLibGenRTMTypedefAndMacros	0.2804032	4.3%	1	0.28040320000	0.0000000	0.0%	//

Analyzing the Report

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 and does not include the time spent in subfunctions called by the function
- Functions are hyperlinks that take you to the details related to that specific function.

A situation where the profiler report may be helpful is 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`
- any type of comment (`%%` or `/% stuff %/`)

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 that were written by you, 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 when they are not really needed. One line functions might be a bottleneck for code generation speed. When readability is not greatly impacted, consider expanding out 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, M-file and Fortran S-functions.

Introduction (p. 7-2)	Finding information about writing S-functions to be used for code generation
Writing Block Target Files to Inline S-Functions (p. 7-3)	Differences between fully-inlined and wrapped S-functions
Inlining C MEX S-Functions (p. 7-5)	Calls made by C S-functions and how to handle them
Inlining M-File S-Functions (p. 7-18)	Accelerating M-file S-function performance
Inlining Fortran (F-MEX) S-Functions (p. 7-20)	How the <code>timestwo</code> function coded in Fortran is handled
TLC Coding Conventions (p. 7-24)	Make your TLC code more robust by observing case conventions and using library functions
Block Target File Methods (p. 7-29)	Descriptions of the functions needed to emit block code
Loop Rolling (p. 7-37)	An example that handles multiple inputs
Error Reporting (p. 7-40)	Help in finding the source of trouble

Introduction

Writing S-functions that will be included in code generated by Real-Time Workshop or Real-Time Workshop Embedded Coder involves requirements that go beyond writing S-functions used only for simulation in Simulink. Before you proceed to inline an S-function you should make sure that it meets these requirements and will function as you expect it to. You therefore might want to read Chapter 10, “Writing S-Functions for Real-Time Workshop” in the Real-Time Workshop documentation if you have not already done so. If your S-function is multirate, you also might want to refer to Chapter 8, “Models with Multiple Sample Rates” in the Real-Time Workshop documentation, and the section “Rate Grouping Compliance and Compatibility Issues” in the Real-Time Workshop Embedded Coder documentation.

Writing Block Target Files to Inline S-Functions

With C MEX S-functions, all targets except ERT will support calling the original C MEX code if the source code (.c file) is available when Real-Time Workshop enters its build phase. For S-functions that are in Fortran or .m, you must inline them in order 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 either make it *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 needed 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, the overhead of the function call must be considered when weighing the option of fully inlining the block algorithm or generating function calls.

If a decision has been made to go with function-based code generation, there are two more options to consider:

- Write all the function(s) once, put them in .c file(s) 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 that in addition to the methods such as `Start` and `Outputs` will also conditionally generate more functions in separate code generation buffers to be written to a separate .c file that contains customized versions of functions (data types, widths, algorithms,

etc.), but only the functions needed 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 2-10 and “Writing Wrapper S-Functions” in the Simulink Writing S-Functions documentation.

Inlining C MEX S-Functions

When a Simulink model contains an S-function and a corresponding TLC block target file exists for that S-function, Real-Time Workshop inlines the S-function. Inlining an S-function can produce more efficient code by eliminating the S-function Application Program Interface (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 seven functions, even if the routine is empty for the particular S-function.

Function	Purpose
<code>mdlInitializeSizes</code>	Initialize the sizes array.
<code>mdlInitializeSampleTimes</code>	Initialize the sample times array.
<code>mdlInitializeConditions</code>	Initialize the states.
<code>mdlOutputs</code>	Compute the outputs.
<code>mdlUpdate</code>	Update discrete states.
<code>mdlDerivatives</code>	Compute the derivatives of continuous states.
<code>mdlTerminate</code>	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 only called 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 directory as the S-function's MEX-file. Then, at build time, the target file is executed instead of setting up function calls into the S-function's `.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 may 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 non-tunable parameters (typically set from checkboxes and popups on a masked S-function) that are written via the `mdlRTW` method of the S-function using `ssWriteRTWParamSettings`. The S-function's 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 the Run-Time Parameters section of the Writing S-functions in the Simulink documentation for more information on how to create and use run-time parameters. Also see the `sfcdemo_runtime` demo in the S-function demos for examples of how to create and use the two classes of parameters. The demo source files, which you can inspect and adapt, are:

- `toolbox/simulink/blocks/tlc_c/sfun_runtime1.c`
- `toolbox/simulink/blocks/tlc_c/sfun_runtime1.tlc`
- `toolbox/simulink/blocks/tlc_c/sfun_runtime2.c`
- `toolbox/simulink/blocks/tlc_c/sfun_runtime2.tlc`
- `toolbox/simulink/blocks/tlc_c/sfun_runtime3.c`
- `toolbox/simulink/blocks/tlc_c/sfun_runtime3.tlc`

A Complete Example

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_sfuns.h"
#endif
```

The following two sections show the difference in the code the Real-Time Workshop generates for `model.c` containing noninlined and inlined versions of S-function `foogain`. The model contained no other Simulink blocks.

For information about how to generate code with the Real-Time Workshop, see the Real-Time Workshop documentation.

Comparison of Noninlined and Inlined Versions of `model.c`

Without a TLC file to define the S-function specifics, the Real-Time Workshop must call the MEX-file S-function through the S-function API. The code below is the `model.c` file for the noninlined S-function (i.e., no corresponding TLC file).

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 */

```

By including this simple target file for this S-function block, the `model.c` code is generated as

```
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 all block target files, and must be the first executable statement in the block target file. This directive guarantees that the Target Language Compiler does not execute an inappropriate target file for S-function `foogain`.
- The input to `foo` is `rtGROUND` (a Real-Time Workshop global equal to 0.0) since `foo` is the only block in the model and its input is unconnected.
- Including a TLC file for `foogain` eliminated the need for an S-function registration segment for `foogain`. This significantly reduces code size.
- The TLC code will inline the gain parameter when Real-Time Workshop 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's 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 all 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 for purposes such as caching 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 `DWorks`. For the smallest effort in writing both the S-function and block target file and for

automatic conformance to both static and malloc instance data on targets such as grt and grt_malloc, The MathWorks recommends using data-typed work vectors when writing S-functions with instance data, accessed with the `ssSetDWork` and `ssGetDWork` methods.

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 a snap, since you have no TLC code to write for allocating and freeing the `DWork` memory — Real-Time Workshop takes care of this for you.

Additionally, if you want to bundle up groups of `DWorks` into structures for passing to functions, you can populate the structure with pointers to `DWork` arrays in both your S-function's `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 `DWorks` 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, i.e., both implementations use `DWorks` in the same way so that the inlined code can be used with the Simulink Accelerator without any changes to the C MEX S-function or the block target file.

Using Inlined Code With the Simulink Accelerator

By default, the Simulink Accelerator will call your C MEX S-function as part of an accelerated model simulation. If you want to instead 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 will not be able to 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 are not used in generating code for an accelerated model.

Inlining M-File S-Functions

All of the functionality of M-file S-functions can be inlined in the generated code. Writing a block target file for an M-file S-function is essentially identical to the process for a C MEX S-function.

Note that while you can fully inline an M-file S-function to achieve top performance—even with Simulink Accelerator—the MATLAB Math Library is not included with Real-Time Workshop, so any high-level MATLAB commands and functions you use in the M-file S-function must be written by hand in the block target file.

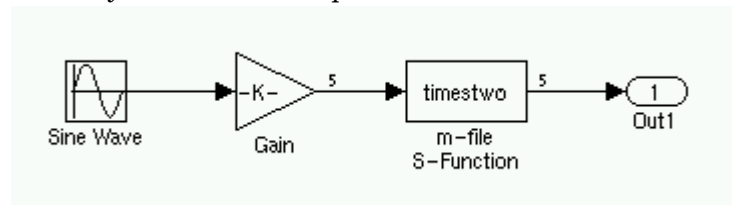
A quick example will illustrate the equivalence of C MEX and M-file S-functions for code generation. The M-file S-function `timestwo.m` is equivalent to the C MEX S-function `timestwo`. In fact, the TLC file for the C MEX S-function `timestwo` will work for the M-file S-function `timestwo.m` as well! Since TLC only requires 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 what the TLC file will be used for

```
%implements "timestwo" "C"
```

To try this out for yourself, copy file `timestwo.m` from `matlabroot/toolbox/simulink/blocks/` to a temporary directory, then copy the file `timestwo.tlc` from `matlabroot/toolbox/simulink/blocks/tlc_c/` to the same temporary directory. In MATLAB, `cd` to the temporary directory and make a Simulink model with an S-function block that calls `timestwo`. Since the MATLAB search path will find `timestwo.m` in the current directory before finding the C MEX S-function `timestwo` in the `matlabpath`, Simulink will use the M-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 M-file S-function `timestwo.m` in the temporary directory. Here is the sample model.



Upon generating code, you will find that the `timestwo.tlc` file was used to inline the M-file S-function with code that looks like this (with an input signal width of 5 in this example):

```

/* S-Function Block: <Root>/m-file 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 was

```

%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 M-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 it show up in the generated code, then work up to algorithmic changes.

Inlining Fortran (F-MEX) S-Functions

The capabilities of Fortran MEX S-functions can be fully inlined using a TLC block target file. With a simple F MEX S-function version of the ubiquitous “timestwo” function, this interface can be illustrated. Here is the sample Fortran S-function code:

```
C
C   FTIMESTWO.FOR
C   $Revision: 1.1$
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=====
C
C       SUBROUTINE SIZES(SIZE)
C       .. Array arguments ..
C       INTEGER*4      SIZE(*)
```

```
C      .. Parameters ..
C      INTEGER*4      NSIZES
C      PARAMETER      (NSIZES=6)

C      SIZE(1) = 0
C      SIZE(2) = 0
C      SIZE(3) = 1
C      SIZE(4) = 1
C      SIZE(5) = 0
C      SIZE(6) = 1

C      RETURN
C      END

C
C=====
C
C      Function:  OUTPUT
C
C      Abstract:
C      Perform output calculations for continuous
C      signals.
C
C=====
C      .. Parameters ..
C      SUBROUTINE OUTPUT(T, X, U, Y)
C      REAL*8      T
C      REAL*8      X(*), U(*), Y(*)

C      Y(1) = U(1) * 2.0

C      RETURN
C      END

C
C=====
C
C      Stubs for unused functions.
C
C=====
```

```
        SUBROUTINE INITCOND(X0)
        REAL*8          X0(*)
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
```

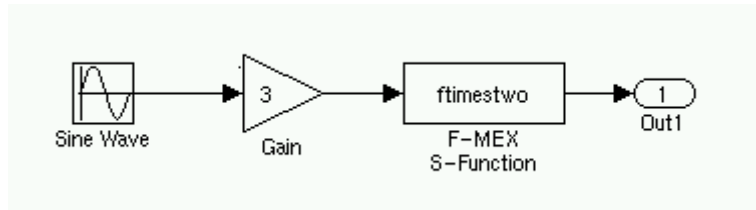
Copy the above code into file `ftimestwo.for` in a convenient working directory.

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 directory along with the file `simulink.for` from `matlabroot/simulink/src/`. This is done with the `mex` command at the MATLAB command prompt:

```
mex -fortran ftimestwo.for simulink.for
```

And now reference this block from a simple Simulink model set with a fixed step solver and the `grt` target.



The TLC code needed to inline this block is a modified form of the now familiar `timestwo.tlc`. In your working directory, 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

These guidelines help ensure that the programming style in each target file is consistent, and hence, more easily modifiable.

Begin Identifiers with Uppercase Letters

All identifiers in the Real-Time Workshop file begin with an uppercase letter. For example,

```
NumModelInputs      1
NumModelOutputs     2
NumNonVirtBlocksInModel 42
DirectFeedthrough   yes
NumContStates       10
```

Block records that contain a Name identifier should start the name with an uppercase letter since the Name identifier is often promoted into the parent scope. For example, a block may contain

```
Block {
  :
  :
  RWork      [4, 0]
  :
  NumRWorkDefines 4
  RWorkDefine {
    Name      "TimeStampA"
    Width     1
    StartIndex 0
  }
}
```

Since 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” Real-Time Workshop identifier. For example,

```
%assign TID = 1
```

produces an error because TID identifier is not qualified by Block. However, a “qualified” assignment does not generate a warning:

```
%assign Block.TID = 1
```

does not generate a warning because the Target Language Compiler assumes the programmer is intentionally modifying an identifier since the assignment contains a qualifier.

Begin Global Variable Assignments with Uppercase Letters

Global TLC variable assignments should start with uppercase letters. A global variable is any 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. In some sense, global assignments have the same scope as Real-Time Workshop 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

Since the Real-Time Workshop 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” on page 8-1.

All Real-Time Workshop global variables start with `rt` and all Real-Time Workshop global functions start with `rt_`.

Avoid naming global variables in your run-time interface modules that start with `rt` or `rt_` since they may conflict with Real-Time Workshop 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.

```

%% Function: Outputs =====
%% Abstract:
%%      Y = U * K
%%
Note c { %function Outputs(block, system) Output
        /* %<Type> Block: %<Name> */ _____ | Note a
        %assign rollVars = ["U", "Y", "P"] _____ | Note e
Notes d, f { %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
              _____ | Note b
        %endfunction

```


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 in order 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 via guard code so that they may be referenced via `%include` multiple times without worrying if they have previously been included. It is recommended that you follow this same practice for any TLC library files that you yourself create.

The convention is to use a variable with the same name as the base filename, uppercased 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 namespaces. For example, when writing TLC code for a block and you are adding any 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

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.

Block Target File Mapping

The *block target file mapping* specifies which target file should be used to generate code for which block type. This mapping resides in `matlabroot/rtw/c/tlc/mw/genmap.tlc`. All the TLC files listed are located in directories within `matlabroot/rtw/c/tlc` for C.

Block Functions

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 any generated code:

- `BlockInstanceSetup(block, system)`
- `BlockTypeSetup(block, system)`

The following functions all generate executable code that Real-Time Workshop places appropriately:

- `Enable(block, system)`
- `Disable(block, system)`
- `Start(block, system)`
- `InitializeConditions(block, system)`
- `Outputs(block, system)`
- `Update(block, system)`
- `Derivatives(block, system)`
- `Terminate(block, system)`

In object-oriented programming terms, these functions are polymorphic in nature since 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" on page 8-1.

BlockInstanceSetup(block, system)

The `BlockInstanceSetup` function executes for all the blocks that have this function defined in their target files in a model. For example, if there are 10 From Workspace blocks in a model, 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 the Reference chapter 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 there are 10 Lookup Table blocks in the model, the BlockTypeSetup function in look_up.tlc is only called one time. Use this function to perform general work for all blocks of a given type.

See ““TLC Function Library Reference” on page 8-1” 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 requiring a #define and two function declarations in the header file, you could define the following function:

```
%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)

Nonvirtual subsystem Enable functions are created 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 into this subsystem Enable function. See `sin_wave.tlc` for an example of the Enable function.

```
%% Function: Enable =====
%% Abstract:
%% Subsystem Enable code is only required for the discrete form
%% of the Sine Block. Setting the boolean to TRUE causes the
%% Output function to re-sync 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 `output.tlc` in `matlabroot/rtw/c/tlc/blocks` for an example of the Disable function.

Start(block, system)

Include a Start function to place code into 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; see `backlash.tlc`) or code that does not need to be reexecuted when the subsystem in which it resides enables. 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 ends up 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 will call 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 enables. You include an InitializeConditions function to execute code that must reexecute when the subsystem in which it resides enables. See `delay.tlc` for an example of the InitializeConditions function. The following code is an example from `ratelim.tlc`:

```

%% 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 %% InitializeConditions

```

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. See `absval.tlc` for an example of the Outputs function:


```

%% 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 either placed into the model's or the subsystem's Update function, depending on whether or not the block resides in a nonvirtual subsystem. See `delay.tlc` for an example of the Update function.

```
%% Function: Update =====
%% Abstract:
%%      X[i] = U[i]
%%
%function Update(block, system) Output
/* %<Type> Block: %<Name> */
%assign stateLoc = (DiscStates[0]) ? "Xd" : "DWork"
%assign rollVars = ["U", %<stateLoc>]
%roll idx = RollRegions, lcv = RollThreshold, block, ...
    "Roller", rollVars
%assign u = LibBlockInputSignal(0, "", lcv, idx)
%assign x = FcnGetState("", lcv, idx, "")
%<x> = %<u>;
%endroll

%endfunction %% Update
```

`FcnGetState` is a function defined locally in `delay.tlc`.

Derivatives(block, system)

Include a Derivatives function when generating code to compute the block's continuous states. Code generated from this function is either placed into 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.

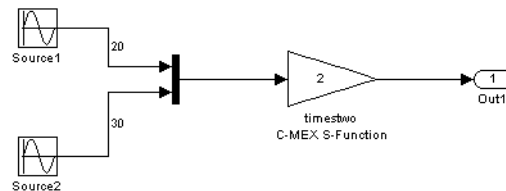
Terminate(block, system)

Include a Terminate function to place any code into `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 a Simulink optimization feature that makes the Mux block *virtual*, meaning that there is no code explicitly generated for the mux and thus no processor cycles 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 snippet 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. Looking at the generated code, we see

```

/* 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 in 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 runtime, but may be worth it if your algorithm performance increases enough to offset the overhead of the buffering.

Loops are generated by using the `%roll` directive. See also `%roll %endroll` on page 5-11 for the reference entry for `%roll` and `%roll` on page 5-31 for a section describing the behavior of `%roll`.

Error Reporting

You may need to detect and report error conditions in your TLC code. Error detection and reporting is needed most often in library functions. While rare, it is also possible to encounter error conditions in block target file code. The reason this is rare, but can occur if there is an unforeseen condition that the S-function `mdlCheckParameters` function does not detect.

To report an error condition detected in your TLC code, use the `LibBlockReportError` or `LibBlockReportFatalError` utility functions. Use of these functions is fully documented in the Reference section. 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 “TLC Error Handling” on page A-1, 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 may change significantly from release to release. A table of obsolete functions and their replacements is shown in *Obsolete Functions*.

Obsolete Functions (p. 8-2)	Deprecated functions and their replacements
Target Language Compiler Functions (p. 8-4)	Function syntax, conventions, and common arguments
Input Signal Functions (p. 8-9)	Functions that process and report on input signals
Output Signal Functions (p. 8-21)	Functions that process and report on output signals
Parameter Functions (p. 8-26)	Functions that process model parameters
Block State and Work Vector Functions (p. 8-31)	Functions that handle storage and states
Block Path and Error Reporting Functions (p. 8-35)	Functions for navigating paths and handling error conditions
Code Configuration Functions (p. 8-37)	Functions for tailoring code elements and comments
Sample Time Functions (p. 8-59)	Functions for handling continuous and discrete time
Other Useful Functions (p. 8-67)	Functions not elsewhere classified
Advanced Functions (p. 8-78)	Functions generally required only for special situations

You can find examples using these functions in `matlabroot/toolbox/simulink/blocks/tlc_c`. The corresponding MEX S-function source code is located in `matlabroot/simulink/src`. M-file S-functions and the MEX-file executables (e.g., `sfunction.dll`) for `matlabroot/simulink/src` are located in `matlabroot/toolbox/simulink/blocks`.

Obsolete Functions

The following table shows obsolete functions and the functions that have replaced them.

Obsolete Function	Equivalent Replacement Function
LibBlockOutputportLocation	LibBlockDstSignalLocation
LibCacheGlobalPrmData	Use the block function Start
LibContinuousState	LibBlockContinuousState
LibControlPortInputSignal	LibBlockSrcSignalLocation
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
LibExternalResetSignal	LibBlockInputSignal
LibIsEqual	Use built-in function ISEQUAL
LibMapSignalSource	FcnMapDataTypedSignalSource
LibMaxBlockIOWidth	Function is not used in the Real-Time Workshop.
LibMaxDataInputPortWidth	Function is not used in the Real-Time Workshop.
LibMaxDataOutputPortWidth	Function is not used in the Real-Time Workshop.
LibPathName	LibGetBlockPath, LibGetFormattedBlockPath
LibPrevZCState	LibBlockPrevZCState

Obsolete Function	Equivalent Replacement Function
LibRenameParameter	Specifying parameter names is now supported via the mdlRTW function in your C-MEX S-function.
LinConvertZCDirection	Function is not used in the Real-Time Workshop.

Target Language Compiler Functions

This section lists the Target Language Compiler functions grouped by category, and provides a description of each function. To view the source code for a function, click on its name.

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 zero. 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 will contain either "", indicating that the current pass through the %roll is being inlined, or it will be 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 of the %roll directive, this is usually specified as "".

Argument	Description
sigIdx or idx	<p data-bbox="568 305 1356 609">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 <code>ucv=""</code>, <code>lcv=""</code>, and <code>sigIdx</code> equal to the desired integer signal index starting at 0. Note, for complex signals, <code>sigIdx</code> can be an overloaded integer index specifying both whether the real or imaginary part is being accessed and which element. When accessing these items inside of a <code>%roll</code>, the <code>sigIdx</code> generated by the <code>%roll</code> directive should be used.</p> <p data-bbox="568 661 1356 730">Most functions that take a <code>sigIdx</code> argument accept it in an overloaded form where <code>sigIdx</code> can be:</p> <ul data-bbox="568 748 1356 1216" style="list-style-type: none"> <li data-bbox="568 748 1356 887">• 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. <li data-bbox="568 895 1356 1069">• An <i>id-num</i> usually of the form (see “Overloading sigIdx” on page 8-6): <ul data-bbox="609 982 1356 1216" style="list-style-type: none"> <li data-bbox="609 982 1356 1069">a <code>"%<tRealPart>%<idx>"</code> (e.g., <code>"re3"</code>). The real part of the signal element. Usually <code>"%<tRealPart>%<sigIdx>"</code> when <code>sigIdx</code> is generated by the <code>%roll</code> directive. <li data-bbox="609 1086 1356 1216">b <code>"%<tImagPart>%<idx>"</code> (e.g., <code>"im3"</code>). The imaginary part of the signal element or <code>" "</code> if the signal is not complex. Usually <code>"%<tImagPart>%<sigIdx>"</code> when <code>sigIdx</code> is generated by the <code>%roll</code> directive. <p data-bbox="568 1234 1356 1269">The <code>idx</code> name is used when referring to a state or work vector.</p> <p data-bbox="568 1286 1356 1564">Functions that accept the three arguments <code>ucv</code>, <code>lcv</code>, <code>sigIdx</code> (or <code>idx</code>) are called differently depending upon whether or not they are used with in a <code>%roll</code> directive. If they are used within a <code>%roll</code> directive, <code>ucv</code> is generally specified as <code>" "</code> and <code>lcv</code> and <code>sigIdx</code> are the same as those specified in the <code>%roll</code> directive. If they are not used with in a <code>%roll</code> directive, <code>ucv</code> and <code>lcv</code> are generally specified as <code>" "</code> and <code>sigIdx</code> specifies which index to access.</p>

Argument	Description
paramIdx	Parameter index. Sometimes referred to as the parameter element index. The handling of this parameter is very similar to sigIdx (i.e., 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 we 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	<code>u0[2].re</code>	<code>real_T</code>
"im3"	yes	Imaginary part of element 3	<code>u0[2].im</code>	<code>real_T</code>
"3"	yes	Complex container of element 3	<code>u0[2]</code>	<code>creal_T</code>
3	yes	Complex container of element 3	<code>u0[2]</code>	<code>creal_T</code>

sigIdx	Complex	Function Returns	Example	Data Type
"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:

1 We 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 i
"3"	yes	Complex container of element i
3	yes	Complex container of element i
"re3"	no	Element i
"im3"	no	" "
"3"	no	Element i
3	no	Element i

Notes

- The vector index is only added for wide signals.
- If ucv is not an empty string (" "), then the 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 this function returns `"&y%<portIdx>[%<lcv>]"` and `sigIdx` is ignored.
- It is assumed here that the roller has appropriately 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

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 appropriate 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, an external input, or it can be a literal constant (e.g, 5.0).

Note Never use this function to access the address of an input signal.

Since 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` may 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 pre-processor) 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 any such situations by using `LibBlockInputSignalAddr`.

```
%assign uAddr = LibBlockInputSignalAddr(0, "", lcv, sigIdx)
x = %<uAddr>;
```

Real-Time Workshop tracks signals and parameters accessed by their address 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 only allowed 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 is appropriate for 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 always the starting index of the current roll region and `lcv` will be `" "` or an indexing variable. The following are examples of valid values:

- Example 1:

```
LibBlockInputSignal(0, "", lcv, sigIdx)    rtB.blockname[0]
```

- Example 2:

```
LibBlockInputSignal(0, "", lcv, sigIdx)    u[i]
```

In Example 1, `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` was non-zero, then `rtB.blockname[sigIdx]` would be returned. For example if `sigIdx` was 3, then `rtB.blockname[3]` would be returned.

In Example 2, `LibBlockInputSignal` returns `u[i]` when the current roll region is above the roll threshold and the input port width is non-scalar (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, suppose we have 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 all 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
```

Since 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 needed 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.

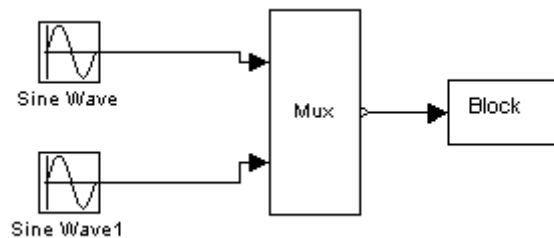
- `rvalue_id[%<ucv>]%<optional_real_or_imag_part>`

`rvalue_id` is obtained by looking at the integer part of `sigIdx`. Specifying `sigIdx` is required 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`. Also, `sigIdx` is used to determine whether the real or imaginary part of a signal is to be accessed.

`optional_real_or_imag_part` is obtained by the string part of `sigIdx` (i.e. "re", or "im", or "").

Note: 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 has no 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, the FlatRoller has no 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 < %<DataInputPort[0].Width>; 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 the MathWorks operates, then you will need to use ucv instead of lcv.

Input Arguments (ucv, lcv, and sigIdx) Handling

Consider the following cases :

Function (case 1, 2, 3,4)	Example Return Value
<code>LibBlockInputSignal(0, "i", "", sigIdx)</code>	<code>rtB.blockname[i]</code>
<code>LibBlockInputSignal(0, "i", "", sigIdx)</code>	<code>rtU.signame[i]</code>
<code>LibBlockInputSignal(0, "", lcv, sigIdx)</code>	<code>u0[i1]</code>
<code>LibBlockInputSignal(0, "", lcv, sigIdx)</code>	<code>rtB.blockname[0]</code>

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` pointing to the external input vector, i.e., the first element that "i" starts with. For example, if you initialize "i" to be starting 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, this function 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 will be 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 this function (inside or outside of a `%roll` directive) will use `sigIdx` when `ucv` and `lcv` are specified as `" "`.

For an example of this function, see `matlabroot/toolbox/simulink/blocks/tlc_c/sfun_multiport.tlc`. See also `matlabroot/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalAddr(portIdx, ucv, lcv, sigIdx)

Returns the appropriate string that provides the memory address of the specified block input port signal.

When you need an input signal address, you must use this function 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). Real-Time Workshop 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, unlike `LibBlockInputSignal()`, the last input argument, `sigIdx`, is not overloaded. 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 function 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` will be `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 will be `real_T`.

```
%assign dtname = LibBlockOutputSignalDataTypeName(0,tRealPart)
```

See function in `matlabroot/rtw/c/tlc/lib/blkio.lib.tlc`.

LibBlockInputSignalConnected(portIdx)

Returns 1 if the specified input port is connected to a block other than the Ground block and 0 otherwise.

See function in `matlabroot/rtw/c/tlc/lib/blkio.lib.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, this function returns the data type of the real part (or the imaginary part) of the signal.

See function in `matlabroot/rtw/c/tlc/lib/blkio.lib.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` will be `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 will be `real_T`.

```
%assign dtname = LibBlockInputSignalDataTypeName(0,tRealPart)
```

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockInputSignalDimensions(portIdx)

Returns the dimensions vector of specified block input port, e.g., [2,3].

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockInputSignalIsComplex(portIdx)

Returns 1 if the specified block input port is complex, 0 otherwise.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockInputSignalIsFrameData(portIdx)

Returns 1 if the specified block input port is frame based, 0 otherwise.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockInputSignalLocalSampleTimeIndex(portIdx)

Returns the local sample time index corresponding to the specified block input port.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockInputSignalNumDimensions(portIdx)

Returns the number of dimensions of the specified block input port.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockInputSignalOffsetTime(portIdx)

Returns the offset time corresponding to the specified block input port.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockInputSignalSampleTime(portIdx)

Returns the sample time corresponding to the specified block input port.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockInputSignalSampleTimeIndex(portIdx)

Returns the sample time index corresponding to the specified block input port.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockInputSignalWidth(portIdx)

Returns the width of the specified block input port index.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockInputPortIndexMode(block, idx)

Purpose

Determines the index mode of a block's input port.

Description

If a block's input port has been set as an index port and its indexing base is marked as zero-based or one-based, this information gets written into the *model.rtw* file. This function queries the indexing base in order to branch to different code according to what the input port indexing base is.

Returns

" " for a non-index port, and "Zero-based" or "One-based" otherwise.

Arguments

block - block record

idx - port index

Example

```
%if LibBlockInputPortIndexMode(block, idx) == "Zero-based"  
    ...  
%elseif LibBlockInputPortIndexMode(block, idx) == "One-based"  
    ...  
%else  
    ...  
%endif
```

See function in *matlabroot/rtw/c/tlc/mw/blocklib.tlc*.

Output Signal Functions

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 the appropriate 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 Never use this function to access the address of an output signal.

Real-Time Workshop 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 function in `matlabroot/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalAddr(portIdx, ucv, lcv, sigIdx)

Returns the appropriate string that provides the memory address of the specified block output port signal.

When an output signal address is needed, you must use this function 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, 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 function in *matlabroot/rtw/c/tlc/lib/blkio.lib.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 name placed in `dtname` will be `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 will be `real_T`.

```
%assign dtname = LibBlockOutputSignalDataTypeName(0,tRealPart)
```

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputSignalBeingMerged(portIdx)

Returns whether the specified output port is connected to a merge block.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputSignalConnected(portIdx)

Returns 1 if the specified output port is connected to a block other than the Ground block and 0 otherwise.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputSignalDataTyped(portIdx)

Returns the numeric ID corresponding to the data type of the specified block output port.

If the output port signal is complex, this function returns the data type of the real (or the imaginary) part of the signal.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.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` will be `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 will be `real_T`.

```
%assign dtname = LibBlockOutputSignalDataTypeName(0,tRealPart)
```

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputSignalDimensions(portIdx)

Returns the dimensions of specified block output port.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputSignalIsComplex(portIdx)

Returns 1 if the specified block output port is complex, 0 otherwise.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputSignalIsFrameData(portIdx)

Returns 1 if the specified block output port is frame based, 0 otherwise.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputSignalLocalSampleTimeIndex(portIdx)

Returns the local sample time index corresponding to the specified block output port.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputSignalNumDimensions(portIdx)

Returns the number of dimensions of the specified block output port.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputSignalOffsetTime(portIdx)

Returns the offset time corresponding to the specified block output port.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputSignalSampleTime(portIdx)

Returns the sample time corresponding to the specified block output port.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputSignalSampleTimeIndex(portIdx)

Returns the sample time index corresponding to the specified block output port.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputSignalWidth(portIdx)

Returns the width of specified block output port.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputPortIndexMode(block, idx)**Purpose**

Determines the index mode of a block's output port.

Description

If a block's output port has been set as an index port and its indexing base is marked as zero-based or one-based, this information gets written into the *model.rtw* file. This function queries the indexing base in order to branch to different code according to what the output port indexing base is.

Returns

" " for a non-index port, and "Zero-based" or "One-based" otherwise.

Arguments

block - block record

idx - port index

Example

```
%if LibBlockOutputPortIndexMode(block, idx) == "Zero-based"  
    ...  
%elseif LibBlockOutputPortIndexMode(block, idx) == "One-based"  
    ...  
%else  
    ...  
%endif
```

See function in *matlabroot/rtw/c/tlc/mw/blocklib.tlc*.

Parameter Functions

LibBlockMatrixParameter(param,rucv,rlcv,ridx,cucv,clcv,cidx)

Returns the appropriate 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 rlcvc 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 function in *matlabroot/rtw/c/tlc/lib/paramlib.tlc*.

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., rlcvc and clcv should both be the empty string).

See function in *matlabroot/rtw/c/tlc/lib/paramlib.tlc*.

LibBlockMatrixParameterBaseAddr(param)

Returns the base address of a matrix parameter.

See function in *matlabroot/rtw/c/tlc/lib/paramlib.tlc*.

LibBlockParameter(param, ucv, lcv, sigIdx)

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, this function returns the appropriate reference to a block parameter.

The returned value is always a valid rvalue (right-side value for an expression). For example,

Case	Function Call	May 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 this function, 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

Case	Rolling	Inline Parameter	Type	Result	Required In Memory
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 true, the parameter must be placed in memory (RAM) since it's accessed inside a for-loop.

Note This function also supports expressions when used with inlined parameters and parameter tuning.

For example, if the parameter field had the MATLAB expression '2*a', this function will return the C expression '(2 * a)'. The list of functions supported by this function 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 nonscalars.

See the Real-Time Workshop documentation about tunable parameters for more details on the exact functions and syntax that is supported.

Warning

Do not use this function to access the address of a parameter, or you may end up referencing a number (i.e., &4.55) when the parameter is inlined. You can avoid this situation by using `LibBlockParameterAddr()`.

See function in `matlabroot/rtw/c/tlc/lib/paramlib.tlc`.

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.

Also, trying to access the address of an expression when inline parameters is on and the expression has multiple tunable/rolled variables in it will result in an error.

See function in `matlabroot/rtw/c/tlc/lib/paramlib.tlc`.

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.

Note that Accessing the address of an expression when **Inline parameters** is on and the expression has multiple tunable/rolled variables in it will result in an error.

See function in `matlabroot/rtw/c/tlc/lib/paramlib.tlc`.

LibBlockParameterDataTypeId(param)

Returns the numeric ID corresponding to the data type of the specified block parameter.

See function in `matlabroot/rtw/c/tlc/lib/paramlib.tlc`.

LibBlockParameterDataTypeName(param, reim)

Returns the name of the data type corresponding to the specified block parameter.

See function in `matlabroot/rtw/c/tlc/lib/paramlib.tlc`.

LibBlockParameterDimensions(param)

Returns a row vector of length N (where $N \geq 1$) 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
```

This function 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 function in *matlabroot/rtw/c/tlc/lib/paramlib.tlc*.

LibBlockParameterIsComplex(param)

Returns 1 if the specified block parameter is complex, 0 otherwise.

See function in *matlabroot/rtw/c/tlc/lib/paramlib.tlc*.

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 function in *matlabroot/rtw/c/tlc/lib/paramlib.tlc*.

LibBlockParameterWidth(param)

Returns the number of elements (width) of a parameter.

See function in *matlabroot/rtw/c/tlc/lib/paramlib.tlc*.

Block State and Work Vector Functions

LibBlockContinuousState(ucv, lcv, idx)

Returns a string corresponding to the specified block continuous state (CSTATE) element.

See function 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 function 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 function in *matlabroot/rtw/c/tlc/lib/blocklib.tlc*.

LibBlockDWork(dwork, ucv, lcv, sigIdx)

Returns a string corresponding to the specified block DWORK element.

Note, the last input argument is overloaded to handle complex DWorks.

`sigIdx = "re3"`— returns the real part of element 3 if the `dwork` is complex, otherwise returns element 3.

`sigIdx = "im3"`— returns the imaginary part of element 3 if the `dwork` is complex, otherwise returns "".

`sigIdx = "3"` — returns the complex container of element 3 if the `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 function 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 function in *matlabroot/rtw/c/tlc/lib/blocklib.tlc*.

LibBlockDWorkDataTypeId(dwork)

Returns the data type ID of specified block `DWORK`.

See function in *matlabroot/rtw/c/tlc/lib/blocklib.tlc*.

LibBlockDWorkDataTypeName(dwork, reim)

Returns the data type name of specified block `DWORK`.

See function in *matlabroot/rtw/c/tlc/lib/blocklib.tlc*.

LibBlockDWorkIsComplex(dwork)

Returns 1 if the specified block `DWORK` is complex, returns 0 otherwise.

See function in *matlabroot/rtw/c/tlc/lib/blocklib.tlc*.

LibBlockDWorkName(dwork)

Returns the name of the specified block `DWORK`.

See function in *matlabroot/rtw/c/tlc/lib/blocklib.tlc*.

LibBlockDWorkStorageClass(dwork)

Returns the storage class of specified block `DWORK`.

See function in *matlabroot/rtw/c/tlc/lib/blocklib.tlc*.

LibBlockDWorkStorageTypeQualifier(dwork)

Returns the storage type qualifier of specified block `DWORK`.

Function: `LibBlockDWorkStorageTypeQualifier(dwork) void`

See function 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 function in `matlabroot/rtw/c/tlc/lib/blocklib.tlc`.

LibBlockDWorkWidth(dwork)

Returns the width of the specified block DWORK.

See function 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 function 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 function in `matlabroot/rtw/c/tlc/lib/blocklib.tlc`.

LibBlockMode(ucv, lcv, idx)

Returns a string corresponding to the specified block MODE element.

See function in `matlabroot/rtw/c/tlc/lib/blocklib.tlc`.

LibBlockNonSampledZC(ucv, lcv, NonSampledZCIdx)

Returns a string corresponding to the specified block NonSampledZC.

`LibBlockNonSampledZC` returns the appropriate element for the non-sampled zero crossing state based on `ucv`, `lcv`, and `NonSampledZCIdx`.

Arguments:

ucv: User control variable string

lcv: Loop control variable string

NonSampledZCIdx: Non-Sampled zero crossing index

See function 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 function 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*, would typically be a symbol defined in the *mdlRTW()* routine of the C MEX file with something like the code below.

```
ssWriteRTWorkVect(..., "RWork", ..., "MyRWorkName", ...)
```

Alternately, if no such RWork defines have been made, *definedrwork* will be ignored and the raw RWork vector will be accessed. In this case all uses in a loop rolling context are disallowed.

See function in *matlabroot/rtw/c/tlc/lib/blocklib.tlc*.

Block Path and Error Reporting Functions

LibBlockReportError(block,errorstring)

This should be used when reporting errors for a block. This function is designed to be used from block target files (e.g., the TLC file for an inlined S-function).

This function 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 = []`. Specifically

```
LibBlockReportError([], "error string")           --If block is scoped
LibBlockReportError(blockrecord, "error string") --If block record is
                                                    available
```

See function in *matlabroot/rtw/c/tlc/lib/utillib.tlc*.

LibBlockReportFatalError(block,errorstring)

This should be used when reporting fatal (assert) errors for a block. Use this function for defensive programming. Refer to “TLC Error Handling” on page A-1.

See function in *matlabroot/rtw/c/tlc/lib/utillib.tlc*.

LibBlockReportWarning(block,warnstring)

This should be used when reporting warnings for a block. This function is designed to be used from block target files (e.g., the TLC file for an inlined S-function).

This function 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 = []`.

Specifically

```
LibBlockReportWarning([], "warn string")         --If block is scoped
LibBlockReportWarning(blockrecord, "warn string") --If block record is
                                                    available
```

See function 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 which may be present in the name.

See function 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 may be present in the name. Currently, the only other special string sequences defined are '/' and '*/'.

The full block path name string is useful when 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 function in *matlabroot/rtw/c/tlc/lib/utillib.tlc*.

LibGetFormattedBlockPath(block)

LibGetFormattedBlockPath returns the full path name string of a block without any special characters. The string returned from this function 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 '/' to distinguish it from a path separator.

Use LibGetBlockPath to get the block path needed by MATLAB functions used in reference blocks in your model.

See function in *matlabroot/rtw/c/tlc/lib/utillib.tlc*.

Code Configuration Functions

LibAddSourceFileCustomSection(file, builtInSection, newSection)

Add 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.

No action if the section already exists, except to report an error if a inconsistent built-in section association is attempted.

Only available with Real-Time Workshop Embedded Coder.

Arguments:

`file` - Source file reference (Scope)

`builtInSection` - Name of the associated built-in section (String)

`newSection` - Name of the new (custom) section (String)

See function in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc`.

LibAddToCommonIncludes(incFileName)

Adds items to a unique-ified list of `#include/package` spec items.

Should be called from block TLC methods to specify generation of `#include` statements in `model.h`. Specify the names of local files bare, e.g., `"myinclude.h"`, but specify the names of files on the include path files inside angle brackets, e.g., `"<sysinclude.h>"`. Each call to this function adds the specified file to the list only if it is not already there. `<math.h>` and `"math.h"` are considered different files for the purpose of uniqueness. The `#include` statements are placed inside `model.h`.

Example:

```
%<LibAddToCommonIncludes("tpu332lib.h")>
```

See function in `matlabroot/rtw/c/tlc/lib/cachelib.tlc`.

LibAddToModelSources(newFile)

This function serves two purposes:

- To notify the Real-Time Workshop build process that it must build with the specified source file, and
- To update the 'SOURCES: file1.c file2.c ...' comment in the generated code.

For inlined S-functions, `LibAddToModelSources` is generally called from `BlockTypeSetup`. This function adds a file name to the list of sources needed to build this model. This function 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.

As an S-function author, we recommend using the `SFunctionModules` block parameter instead of this function. See [Writing S-functions](#).

See function in `matlabroot/rtw/c/tlc/lib/utillib.tlc`.

LibCacheDefine(buffer)

Each call to this function appends your buffer to the existing cache buffer. For blocks, this function is generally called from `BlockTypeSetup`.

This function caches `#define` statements for inclusion in `model.h` (or `model_private.h`). `LibCacheDefine` should be called from inside `BlockTypeSetup` to cache a `#define` statement. Each call to this function 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 function 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 this function appends your buffer to the existing cache buffer. The extern statements are placed in `model.h`.

Example

```
%openfile buffer
extern real_T mydata;
%closefile buffer
%<LibCacheExtern(buffer)>
```

See function 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 this function appends your buffer to the existing cache buffer. The prototypes are placed inside model.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 function in matlabroot/rtw/c/tlc/lib/cachelib.tlc.

LibCacheIncludes(buffer)

LibCacheIncludes should be called from inside BlockTypeSetup to cache #include statements. Each call to this function appends your buffer to the existing cache buffer. The #include statements are placed inside model.h.

Example

```
%openfile buffer
#include "myfile.h"
%closefile buffer
%<LibCacheIncludes(buffer)>
```

See function in matlabroot/rtw/c/tlc/lib/cachelib.tlc.

LibCacheTypedefs(buffer)

LibCacheTypedefs should be called from inside BlockTypeSetup to cache typedef declarations. Each call to this function 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 function in matlabroot/rtw/c/tlc/lib/cachelib.tlc.

LibRegisterGNUMathFcnPrototypes()

Example of registering target-specific math functions. This one registers GNU C math function mappings for a target with a GNU C compiler (e.g., gcc 2.9x.yy+ is compliant).

See function in matlabroot/rtw/c/tlc/lib/mathlib.tlc.

LibRegisterISOCMathFcnPrototypes()

Example of registering target-specific math functions. This function registers ISO C math function mappings for a target with an ISO C 9x compliant compiler (e.g., gcc 2.9x.yy+ is).

See function in matlabroot/rtw/c/tlc/lib/mathlib.tlc.

LibRegisterMathFcnPrototype(RTWName, RTWType, IsExprOK, IsCplx, NumInputs, FcnName, FcnType, HdrFile)

Set a specific name and input prototype of a given function for the current target. This overrides the default names. Data types are in string form.

See function in matlabroot/rtw/c/tlc/lib/mathlib.tlc.

LibCallModelInitialize()

Returns necessary code for calling the model's initialize function (valid for ERT only).

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibCallModelStep(tid)

Returns necessary code for calling the model's step function (valid for ERT only).

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibCallModelTerminate()

Returns necessary code for calling the model's terminate function (valid for ERT only).

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibCallSetEventForThisBaseStep(buffername)

Returns necessary 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 function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibCreateSourceFile(type,creator,name)

Create a new C file, and return its reference. If the file already exists, simply return its 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's 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 function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibGetMdlPrvHdrBaseName()

Return the base name of the model's private header (e.g., *model_private.h*) file

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibGetMdlPubHdrBaseName()

Return the base name of the model's public header (e.g., *model.h*) file

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibGetMdlSrcBaseName()

Return the base name of the model's main source (e.g., *model.c*) file

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibGetModelDotCFile()

Get the record for the *model.c* file. Additional code can then be cached using `LibSetSourceFileSection()`.

Call syntax

```
%assign srcFile = LibGetModelDotCFile()  
%<LibSetSourceFileSection(srcFile, "Functions", mybuf)>
```

Returns

Returns the *model.c* source file record.

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibGetModelDotHFile()

Get the record for the *model.h* file. Additional code can then be cached using `LibSetSourceFileSection()`.

Call syntax

```
%assign hdrFile = LibGetModelDotHFile()  
%<LibSetSourceFileSection(hdrFile, "Functions", mybuf)>
```

Returns

Returns the *model.h* source file record.

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibGetModelName()

Return name of the model (no extension)

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibGetNumSourceFiles()

Get the number of source files (.c and .h) that have been created.

Call syntax

```
%assign numFiles = LibGetNumSourceFiles()
```

Returns

Returns the number of files (Number).

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibGetRTModelErrorStatus()

Returns the code required to get the model error status

Call syntax

```
%<LibGetRTModelErrorStatus(>;
```

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibGetSourceFileCustomSection(file,attrib)

Get a custom section previously created with `LibAddSourceFileCustomSection`.

Arguments:

`file` - Source file reference or index (Scope or Number)

`attrib` - Name of custom section (String)

See function in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc`.

LibGetSourceFileFromIdx(fileIdx)

Return a model file reference based on its index. This is very useful for a common operation on all files. For example, to set the leading file banner of all files.

Call syntax

```
%assign fileH = LibGetSourceFileFromIdx(fileIdx)
```

Argument

`fileIdx` (Number) - Index of model file (that is internally managed by RTW).

Returns

Reference (Scope) to the model file.

See function 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.

Call syntax

```
%assign tag = LibGetSourceFileTag(fileIdx)
```

Argument

`fileIndex` (Number) - File index.

Returns

Returns the tag (String).

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibMdlStartCustomCode(buffer, location)

Place 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 to append to internal cache buffer

location:

"header" to place buffer at top of function

"declaration" same as specifying header

"execution" to place buffer at top of function, but after header

"trailer" to place buffer at bottom of function)

Returns

Nothing

Description

LibMdlStartCustomCode places declaration statements and executable code inside the start function. This code gets output into the following functions depending on the current code format:

Function Name	Code Format
<model>_initialize	Embedded-C
mdlStart	S-function
MdlStart	RealTime, RealTimeMalloc

Each call to this function appends your buffer to the internal cache buffer.
See function in *matlabroot/rtw/c/tlc/mw/hookslib.tlc*.

LibMdlTerminateCustomCode(buffer, location)

Purpose

Place declaration statements and executable code inside the terminate function.

Syntax

```
LibMdlTerminateCustomCode(buffer, location)
```

Arguments

buffer - String buffer to append to internal cache buffer

location -Where to place the buffer's contents

"header"	to place buffer at top of function
"declaration"	same as specifying header
"execution"	to place buffer at top of function, but after header
"trailer"	to place buffer at bottom of function)

Returns

Nothing

Description

LibMdlTerminateCustomCode places declaration statements and executable code inside the terminate function.

This code gets output into the following functions depending on the current code format:

Function Name	Code Format
<i>model_terminate</i>	Embedded-C
<i>mdlTerminate</i>	S-function
<i>MdlTerminate</i>	RealTime, RealTimeMalloc

Each call to this function appends your buffer to the internal cache buffer.

See function in *matlabroot/rtw/c/tlc/mw/hookslib.tlc*.

LibSetRTModelErrorStatus(str)

Returns the code required set the model error status

Argument

str (string) - char * to a C string

Call syntax

```
%<LibSetRTModelErrorStatus("\ "Overrun\ " )>;
```

See function 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 RTW GUI. This function allows you to change the template for a file. Uses the "Code templates" entered into the RTW Templates UI.

Note Custom templates is a feature of RTW Embedded Coder.

Call syntax

```
%assign tag = LibSetSourceFileCodeTemplate(opFile,name)
```

Arguments

opFile (Scope) - Reference to file

name (String) - Name of the desired template

Returns

Nothing

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibSetSourceFileCustomSection(file,attrib,value)

Set a custom section previously created with `LibAddSourceFileCustomSection`. Only available with Real-Time Workshop Embedded Coder.

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 function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibSetSourceFileOutputDirectory(opFile,name)

By default, *.c and *.h files are generated into the RTW build directory. This function allows you to change the default location. Note that the caller is responsible for specifying a valid directory.

Call syntax

```
%assign tag = LibSetSourceFileOutputDirectory(opFile,dirName)
```

Arguments

opFile (Scope) - Reference to file

dirName (String) - Name of the desired output directory

Returns

Nothing

See function in `matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc`.

LibSetSourceFileSection(fileH, section, value)

Add to the contents of a file. Valid file sections include:

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 only depend on intrinsic C types.
PrimitiveTypedefs	Append to the primitive typedef section. Primitive typedefs are those that only depend on intrinsic C types and any typedefs previously defined in the IntrinsicTypes section.
UserTop	Append to the "user top" section.
Typedefs	Append to the typedef section. Typedefs can depend on any previously defined type.
Enums	Append to the enumerated types section.
Definitions	Append to the data definition section.
ExternData	(reserved) RTW extern data.
ExternFcns	(reserved) RTW extern functions.
FcnPrototypes	(reserved) RTW function prototypes.
Declarations	Append to the data declaration section.
Functions	Append to the C functions section.
CompilerErrors	Append to the #warning section.
CompilerWarnings	Append to the #error section.

Documentation	Append to the documentation (comment) section.
UserBottom	Append to the "user bottom" section.

Code is emitted by Real-Time Workshop in the order in which it is listed above.

Call Syntax

Example (iterating over all 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 F00 5.0\n")
```

Arguments

`fileH` - Reference or index to a file (Scope or Number).

`section` - File section of interest (String).

`value` - Value (String).

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibSystemDerivativeCustomCode(system, buffer, location)

Purpose

Place declaration statements and executable code inside the system's derivative function.

Syntax

```
LibSystemDerivativeCustomCode(system, buffer, location)
```


Arguments

`system` - Reference to a system

`buffer` - (String) buffer to append to internal cache buffer

`location` - (String) Where to place the buffer

"header"	to place buffer at top of function
"declaration"	same as specifying header
"execution"	to place buffer at top of function, but after header
"trailer"	to place buffer at bottom of function)

Returns

Nothing

Description

`LibSystemDerivativeCustomCode` places declaration statements and executable code inside the system's derivative function.

This code gets output into the following functions depending on the current code format:

Function Name	Code Format
<code>mdlDerivatives</code>	S-function
<code>MdlDerivatives</code>	RealTime, RealTimeMalloc

This function is not relevant for the Embedded-C code format since blocks with continuous states cannot be used.

Each call to this function 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 any continuous states.

See function in `matlabroot/rtw/c/tlc/mw/hookslib.tlc`.

LibSystemDisableCustomCode(system, buffer, location)

Purpose

Place declaration statements and executable code inside the system's disable function.

Syntax

```
LibSystemDisableCustomCode(system, buffer, location)
```

Arguments

system - Reference to a system

buffer - (String) buffer to append to internal cache buffer

location - (String) Where to place the buffer

"header"	to place buffer at top of function
"declaration"	same as specifying header
"execution"	to place buffer at top of function, but after header
"trailer"	to place buffer at bottom of function)

Returns

Nothing

Description

LibSystemDisableCustomCode places declaration statements and executable code inside the system's disable function. Each call to this function 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 function in *matlabroot/rtw/c/tlc/mw/hookslib.tlc*.

LibSystemEnableCustomCode(system, buffer, location)

Purpose

Place declaration statements and executable code inside the system's enable function.

Syntax

```
LibSystemEnableCustomCode(system, buffer, location)
```

Arguments

system - Reference to a system

buffer - (String) buffer to append to internal cache buffer

location - (String) Where to place the buffer

"header"	to place buffer at top of function
"declaration"	same as specifying header
"execution"	to place buffer at top of function, but after header
"trailer"	to place buffer at bottom of function)

Returns

Nothing

Description

LibSystemEnableCustomCode places declaration statements and executable code inside the system's enable function. Each call to this function 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 function in *matlabroot/rtw/c/tlc/mw/hookslib.tlc*.

LibSystemInitializeCustomCode(system, buffer, location)

Purpose

Place declaration statements and executable code inside the system's initialize function.

Syntax

```
LibSystemInitializeCustomCode(system, buffer, location)
```

Arguments

system - Reference to a system

buffer - (String) buffer to append to internal cache buffer

location - (String) Where to place the buffer

"header"	to place buffer at top of function
"declaration"	same as specifying header
"execution"	to place buffer at top of function, but after header
"trailer"	to place buffer at bottom of function)

Returns

Nothing

Description

LibSystemInitializeCustomCode places declaration statements and executable inside the system's initialize function.

This code gets output into the following functions for the root system depending on the current code format:

Function Name	Code Format
<i>model_initialize</i>	Embedded-C
<i>mdlInitializeConditions</i>	S-function
<i>MdlStart</i>	RealTime, RealTimeMalloc

Code for a subsystem gets output into the subsystem's initialization function. Each call to this function appends your buffer to the internal cache buffer.

Note Enable systems which are not configured to reset on enable get inlined into *MdlStart*. For this case, the system's custom code is found in *MdlStart* above and below the enable system's initialization code.

See function in *matlabroot/rtw/c/tlc/mw/hookslib.tlc*.

LibSystemOutputCustomCode(system, buffer, location)

Purpose

Place declaration statements and executable code inside the system's output function.

Syntax

```
LibSystemOutputCustomCode(system, buffer, location)
```

Arguments

system - Reference to a system

buffer - (String) buffer to append to internal cache buffer

location - (String) Where to place the buffer

"header"	to place buffer at top of function
"declaration"	same as specifying header
"execution"	to place buffer at top of function, but after header
"trailer"	to place buffer at bottom of function)

Returns

Nothing

Description:

LibSystemOutputCustomCode places declaration statements and executable code inside the system's output function. This code gets output into the following functions depending on the current code format:

Function Name	Code Format
<i>model_step</i>	Embedded-C (CombineOutputUpdateFcns is one)
<i>model_output</i>	Embedded-C (CombineOutputUpdateFcns is zero)
mdlOutputs	S-function
MdlOutputs	RealTime, RealTimeMalloc

Each call to this function appends your buffer to the internal cache buffer.

See function in *matlabroot/rtw/c/tlc/mw/hookslib.tlc*.

LibSystemUpdateCustomCode(system, buffer, location)

Purpose

Place declaration statements and executable code inside the system's update function.

Syntax

```
LibSystemUpdateCustomCode(system, buffer, location)
```

Arguments

`system` - Reference to a system

`buffer` - (String) buffer to append to internal cache buffer

`location` - (String) Where to place the buffer

"header"	to place buffer at top of function
"declaration"	same as specifying header
"execution"	to place buffer at top of function, but after header
"trailer"	to place buffer at bottom of function)

Returns

Nothing

Description

`LibSystemUpdateCustomCode` places declaration statements and executable code inside the system's update function. This code gets output into the following functions depending on the current code format:

Function Name	Code Format
<code>model_step</code>	Embedded-C (CombineOutputUpdateFcns is one)
<code>model_update</code>	Embedded-C (CombineOutputUpdateFcns is zero)
<code>mdlUpdate</code>	S-function
<code>MdlUpdate</code>	RealTime, RealTimeMalloc

Each call to this function appends your buffer to the internal cache buffer.

See function in `matlabroot/rtw/c/tlc/mw/hookslib.tlc`.

LibWriteModelData()

Returns necessary data for the model (valid for ERT only).

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibWriteModelInput(tid,rollThreshold)

Return the code necessary to write to a particular root input (i.e., a model inport block). Valid for ERT only.

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 function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibWriteModelInputs()

Return the code necessary to write to root inputs (i.e., all the model inport blocks). Valid for ERT only.

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibWriteModelOutput(tid,rollThreshold)

Return the code necessary to write to a particular root output (i.e., a model outport block). Valid for ERT only.

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 function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibWriteModelOutputs()

Return the code necessary to write to root outputs (i.e., all the model outport blocks). Valid for ERT only.

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

Sample Time Functions

LibAsynchronousTriggeredTID(tid)

Returns whether this TID corresponds to an asynchronous triggered rate.

See function in *matlabroot/rtw/c/tlc/lib/utillib.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 (a real number greater than 0.)
Continuous	0.0
Triggered	-1.0
Constant	-2.0

See function 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 function in *matlabroot/rtw/c/tlc/lib/utillib.tlc*.

LibGetClockTickDataTypeId(tid)

Returns clock tick data type id.

See function in *matlabroot/rtw/c/tlc/lib/utillib.tlc*.

LibGetClockTickHigh(tid)

Return the high byte of integer task time

Returns high order word of integer task time. This function is used when `uint32` pairs are used to store absolute time. The resolution of clock tick can be obtained from `LibGetClockTickStepSize(tid)`.

See function in `matlabroot/rtw/c/tlc/lib/utllib.tlc`.

LibGetClockTickStepSize(tid)

Returns clock tick step size, which is the resolution of the integer task time. This function cannot be used if the task doesn't have a timer.

See function in `matlabroot/rtw/c/tlc/lib/utllib.tlc`.

LibGetElapseTime(system)

Returns time elapsed since the last time the subsystem started to execute.

See function in `matlabroot/rtw/c/tlc/lib/utllib.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 integer elapsed time counter by calling `LibGetElapseTimeCounterDTypeId(system)`.

See function in `matlabroot/rtw/c/tlc/lib/utllib.tlc`.

LibGetElapseTimeCounterDTypeId(system)

Returns the date type id of the integer elapsed time returned by `LibGetElapseTimeCounter`

See function in `matlabroot/rtw/c/tlc/lib/utllib.tlc`.

LibGetElapseTimeResolution(system)

Returns the resolution of the elapsed time returned by LibGetElapseTimeCounter

See function 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. This function allows you to use one function to determine a global TID, independent of port- or block-based sample times.

Calling this function with an integer argument is equivalent to the statement `SampleTimesToSet[sfcnTID][1]`. `SampleTimesToSet` is a matrix that maps local S-function TIDs to global TIDs.

The input argument to this function should be either

- `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.
- or `sfcnTID`: string of the form "InputPortIdxI", "OutputPortIdxI" where I is a number ranging from 0 to the number of ports (e.g., "InputPortIdx0", "OutputPortIdx7"). 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

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 function in *matlabroot/rtw/c/tlc/lib/utillib.tlc*.

LibGetNumSFcnSampleTimes(block)

Returns the number of S-function sample times for a block.

See function in *matlabroot/rtw/c/tlc/lib/utillib.tlc*.

LibGetSfcnTIDType(sfcnTID)

Returns the type of the specified S-function's task identifier (sfcnTID).

- "continuous" if the specified sfcnTID is continuous.
- "discrete" if the specified sfcnTID is discrete.
- "triggered" if the specified sfcnTID is triggered.
- "constant" if 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 function in *matlabroot/rtw/c/tlc/lib/utillib.tlc*.

LibGetTaskTime(tid)

This functions returns the string "ssGetTaskTime(S, tid)" otherwise.

See function in *matlabroot/rtw/c/tlc/lib/utillib.tlc*.

LibGetTaskTimeFromTID(block)

If Code Format is not Embedded-C, this functions returns the string "RTMGet("T")"

if the block is constant or the system is single rate and

"RTMGetTaskTimeForTID(tid)" otherwise.

If Code Format is Embedded-C, this function return

"RTMGetTaskTimeForTID(tid)"

In both cases, S is the name of the SimStruct.

See function in *matlabroot/rtw/c/tlc/lib/utillib.tlc*.

LibIsContinuous(TID)

Returns 1 if the specified task identifier (TID) is continuous, 0 otherwise. Note, TIDs equal to "triggered" or "constant" are not continuous.

See function in *matlabroot/rtw/c/tlc/lib/utllib.tlc*.

LibIsDiscrete(TID)

Returns 1 if the specified task identifier (TID) is discrete, 0 otherwise. Note, task identifiers equal to "triggered" or "constant" are not discrete.

See function in *matlabroot/rtw/c/tlc/lib/utllib.tlc*.

LibIsSFcnSampleHit(sfcnTID)

Returns 1 if a sample hit occurs for the specified local S-function task identifier (TID), 0 otherwise.

The input argument to this function should be either

- `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.
- or `sfcnTID`: "InputPortIdxI", "OutputPortIdxI" (e.g., "InputPortIdx0", "OutputPortIdx7")

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 4 block sample times. The call `LibIsSFcnSampleHit(2)` will return 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 function in *matlabroot/rtw/c/tlc/lib/utllib.tlc*.

LibIsSFcnSingleRate(block)

LibIsSFcnSingleRate returns a boolean value (1 or 0) indicating whether the S-function is single rate (one sample time) or multirate (multiple sample times).

See function 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. This function determines the global TID from the S-function TID and calls LibIsSpecialSampleHit using the global TIDs for both the sample time index (sti) and the task ID (tid).

The input arguments to this function 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 will be 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 function in *matlabroot/rtw/c/tlc/lib/utllib.tlc*.

LibIsSingleRateModel()

Return true if model is single rate and false otherwise.

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibNumAsynchronousSampleTimes()

Return the number of discrete sample times in the model.

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibNumDiscreteSampleTimes()

Return the number of discrete sample times in the model

See function in *matlabroot/rtw/c/tlc/mw/codetemplatelib.tlc*.

LibPortBasedSampleTimeBlockIsTriggered(block)

Determines if the port-based S-function block is triggered.

See function 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 this function in their output functions.

See function in *matlabroot/rtw/c/tlc/lib/blocklib.tlc*.

LibTriggeredTID(tid)

Returns whether this TID corresponds to a triggered rate.

See function in *matlabroot/rtw/c/tlc/lib/utllib.tlc*.

Other Useful Functions

LibBlockExecuteFcnCall(sfcnBlock, callIdx)

For use by inlined S-Functions with function call outputs. Calls LibExecuteFcnCall but provides a simplified argument list. See LibExecuteFcnCall for more information.

Example

```
%foreach callIdx = NumSFcnSysOutputCalls
    %if LibIsEqual(SFcnSystemOutputCall[callIdx].BlockToCall,...
        "unconnected")
        %continue
    %endif
    %% call the downstream system
    %<LibBlockExecuteFcnCall(block, callIdx)>\
%endforeach
```

Returns a string to either call function-call subsystem with the appropriate number of arguments or the generate the subsystem's code right there (inlined).

See function in matlabroot/rtw/c/tlc/lib/asynclib.tlc.

LibCallFCSS(system, simObject, portEl, tidVal)

For use by inlined S-functions with function call outputs. Returns a string to call the function-call subsystem with the appropriate number of arguments or generates the subsystem's code in place (inlined).

Note An S-function can execute a function-call subsystem only via its first output port.

See the SFcnSystemOutputCall record in the model.rtw file.

The return string is determined by the current code format.

Example

```
%foreach fcnCallIdx = NumSFcnSysOutputCalls
```

```
%% call the downstream system
%with SFcnSystemOutputCall[fcnCallIdx]
    %% skip unconnected function call outputs
    %if ISEQUAL(BlockToCall, "unconnected")
        %continue
    %endif
    %assign sysIdx = BlockToCall[0]
    %assign blkIdx = BlockToCall[1]
    %assign ssBlock = System[sysIdx].Block[blkIdx]
    %assign sysToCall = System[ssBlock.ParamSettings.SystemIdx]
    %<LibCallFCSS(sysToCall, tSimStruct, FcnPortElement, ...
        ParamSettings.SampleTimesToSet[0][1])>\
    %endwith
%endforeach
```

BlockToCall and FcnPortElement are elements of the SFcnSystemOutputCall record. System is a record within the global CompiledModel record.

This example is from the file
matlabroot/toolbox/simulink/blocks/tlc_c/fncallgen.tlc.

See function in *matlabroot/rtw/c/tlc/lib/syslib.tlc*.

LibDisableFCSS(system, simObject, portEl, tidVal)

For use by inlined S-Functions with function call outputs. Returns a string to either call function-call subsystem with the appropriate number of arguments or the generate the subsystem's code right there (inlined).

Note Used by inlined S-functions to make a function-call, LibCallFCSS returns the call to the function-call subsystem with the appropriate number of arguments or the inlined code. An S-function can execute a function-call subsystem only via its first output port.

See the SFcnSystemOutputCall record in the *model.rtw* file.

The return string is determined by the current code format.

Example

```

%foreach fcnCallIdx = NumSFcnSysOutputCalls
%% call the downstream system
  %with SFcnSystemOutputCall[fcnCallIdx]
%% skip unconnected function call outputs
  %if ISEQUAL(BlockToCall, "unconnected")
    %continue
  %endif
  %assign sysIdx = BlockToCall[0]
  %assign blkIdx = BlockToCall[1]
  %assign ssBlock = System[sysIdx].Block[blkIdx]
  %assign sysToCall = System[ssBlock.ParamSettings.SystemIdx]
  %<LibCallFCSS(sysToCall, tSimStruct, FcnPortElement, ...
  ParamSettings.SampleTimesToSet[0][1])>\
  %endwith
%endforeach

```

BlockToCall and FcnPortElement are elements of the SFcnSystemOutputCall record.

System is a record within the global CompiledModel record.

See function in *matlabroot/rtw/c/tlc/lib/syslib.tlc*.

LibEnableFCSS(system, simObject, portEl, tidVal)

For use by inlined S-Functions with function call outputs. Returns a string to either call function-call subsystem with the appropriate number of arguments or the generate the subsystem's code right there (inlined).

Note Used by inlined S-functions to make a function-call, LibCallFCSS returns the call to the function-call subsystem with the appropriate number of arguments or the inlined code. An S-function can execute a function-call subsystem only via its first output port.

See the SFcnSystemOutputCall record in the *model.rtw* file. The return string is determined by the current code format.

Example

```
%foreach fcnCallIdx = NumSFcnSysOutputCalls
%% call the downstream system
  %with SFcnSystemOutputCall[fcnCallIdx]
%% skip unconnected function call outputs
  %if ISEQUAL(BlockToCall, "unconnected")
  %continue
  %endif
  %assign sysIdx = BlockToCall[0]
  %assign blkIdx = BlockToCall[1]
  %assign ssBlock = System[sysIdx].Block[blkIdx]
  %assign sysToCall = System[ssBlock.ParamSettings.SystemIdx]
  %<LibCallFCSS(sysToCall, tSimStruct, FcnPortElement, ...
    ParamSettings.SampleTimesToSet[0][1])>\
  %endwith
%endforeach
```

BlockToCall and FcnPortElement are elements of the SFcnSystemOutputCall record. System is a record within the global CompiledModel record.

See function in *matlabroot/rtw/c/tlc/lib/syslib.tlc*.

LibExecuteFcnCall(ssBlock, portEl, tidVal)

For use by inlined S-Functions with function call outputs. Returns a string to either call function-call subsystem with the appropriate number of arguments or the generate the subsystem's code right there (inlined).

Example

```
%foreach fcnCallIdx = NumSFcnSysOutputCalls
%% call the downstream system
  %with SFcnSystemOutputCall[fcnCallIdx]
%% skip unconnected function call outputs
  %if ISEQUAL(BlockToCall, "unconnected")
  %continue
  %endif
  %assign sysIdx = BlockToCall[0]
  %assign blkIdx = BlockToCall[1]
  %assign ssBlock = System[sysIdx].Block[blkIdx]
  %<LibExecuteFcnCall(ssBlock, FcnPortElement, ...
```

```

ParamSettings.SampleTimesToSet[0][1])>\
    %endwith
%endforeach

```

BlockToCall and FcnPortElement are elements of the SFCnSystemOutputCall record.

This example is from the file:

matlabroot/toolbox/simulink/blocks/tlc_c/fncallgen.tlc

See function in *matlabroot/rtw/c/tlc/lib/syslib.tlc*.

LibExecuteFcnDisable(ssBlock, portEl, tidVal)

For use by inlined S-Functions with function call outputs. Returns a string to either call function-call subsystem with the appropriate number of arguments or the generate the subsystem's code right there (inlined).

Example

```

%foreach fcnCallIdx = NumSFCnSysOutputCalls
%% call the downstream system
    %with SFCnSystemOutputCall[fcnCallIdx]
%% skip unconnected function call outputs
    %if ISEQUAL(BlockToCall, "unconnected")
        %continue
    %endif
    %assign sysIdx = BlockToCall[0]
    %assign blkIdx = BlockToCall[1]
    %assign ssBlock = System[sysIdx].Block[blkIdx]
    %<LibExecuteFcnCall(ssBlock, FcnPortElement, ...
        ParamSettings.SampleTimesToSet[0][1])>\
    %endwith
%endforeach

```

BlockToCall and FcnPortElement are elements of the SFCnSystemOutputCall record.

This example is from the file:

matlabroot/toolbox/simulink/blocks/tlc_c/fncallgen.tlc

See function in *matlabroot/rtw/c/tlc/lib/syslib.tlc*.

LibExecuteFcnEnable(ssBlock, portEl, tidVal)

For use by inlined S-Functions with function call outputs. Returns a string to either call function-call subsystem with the appropriate number of arguments or the generate the subsystem's code right there (inlined).

Example

```
%foreach fcnCallIdx = NumSFcnSysOutputCalls
%% call the downstream system
  %with SFcnSystemOutputCall[fcnCallIdx]
%% skip unconnected function call outputs
  %if ISEQUAL(BlockToCall, "unconnected")
    %continue
  %endif
  %assign sysIdx = BlockToCall[0]
  %assign blkIdx = BlockToCall[1]
  %assign ssBlock = System[sysIdx].Block[blkIdx]
  %<LibExecuteFcnCall(ssBlock, FcnPortElement, ...
    ParamSettings.SampleTimesToSet[0][1])>\
  %endwith
%endforeach
```

BlockToCall and FcnPortElement are elements of the SFcnSystemOutputCall record.

This example is from the file:

matlabroot/toolbox/simulink/blocks/tlc_c/fncallgen.tlc

See function in *matlabroot/rtw/c/tlc/lib/syslib.tlc*.

LibGenConstVectWithInit(data, typeId, varId)

Return 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 function in `matlabroot/rtw/c/tlc/lib/utillib.tlc`.

LibGetBlockAttribute(block,attr)

Get a field value inside a Block record.

Call syntax

```
%if LibIsEqual(LibGetBlockAttribute(ssBlock,"MaskType"), ...
    "Task Block")
    %assign isTaskBlock = 1
%endif
```

Returns

Returns the value of the attribute(field) or empty string if it doesn't exist.

See function in `matlabroot/rtw/c/tlc/lib/asynclib.tlc`.

LibGetCallerClockTickCounter(sfcnBlock)

For use by Async S-Functions with function call outputs. Asynchronous tasks can manage their own time. This function is used to access a upstream asynchronous task's time counter. This is preferred when being driven by another asynchronous rate (e.g. Interrupt Block driving a Task block) as the time the interrupt occurred will be used as apposed to the time the Task is allowed to run.

Example

```
%if LibNeedAsyncCounter(block,0)
    /* Use the upstream clock tick counter for this Task. */
    %<LibSetAsyncCounter(block,0, ...
        LibGetCallerClockTickCounter(block))>\
%endif
```

Returns

Returns a string for the counter variable for the upstream asynchronous task.

See function in `matlabroot/rtw/c/tlc/lib/asynclib.tlc`.

LibGetDataComplexNameFromId(id)

Returns the name of the complex data type corresponding to a data type ID. For example, if `id == tSS_DOUBLE` then this function returns `"creal_T"`.

See function in `matlabroot/rtw/c/tlc/lib/dtypelib.tlc`.

LibGetDataEnumFromId(id)

Returns the data type enum corresponding to a data type ID. For example `id == tSS_DOUBLE => enum = "SS_DOUBLE"`. If `id` does not correspond to a built-in data type, this function returns `" "`.

See function in `matlabroot/rtw/c/tlc/lib/dtypelib.tlc`.

LibGetDataTypeIdAliasedThruToFromId(id)

Return the data type `IdAliasedThruTo` corresponding to a data type ID.

See function in `matlabroot/rtw/c/tlc/dtypelib.tlc`.

LibGetDataTypeIdAliasedToFromId(id)

Return the data type `IdAliasedTo` corresponding to a data type ID.

See function in `matlabroot/rtw/c/tlc/lib/dtypelib.tlc`.

LibGetDataTypeIdResolvesToFromId(id)

Return the data type `IdResolvesTo` corresponding to a data type ID.

See function in `matlabroot/rtw/c/tlc/lib/dtypelib.tlc`.

LibGetDataTypeNameFromId(id)

Returns the data type name corresponding to a data type ID.

See function in `matlabroot/rtw/c/tlc/lib/dtypelib.tlc`.

LibGetDataStorageIdFromId(id)

Return the data type `StorageId` corresponding to a data type ID.

See function in `matlabroot/rtw/c/tlc/dtypelib.tlc`.

LibGetT()

Return a string to access the absolute time. You should only use this function to access time.

This function is the TLC version of the SimStruct macro: `ssGetT`.

See function in `matlabroot/rtw/c/tlc/utillib.tlc`.

LibsMajorTimeStep()

Returns a string to access whether the current simulation step is a major time step.

This function is the TLC version of the SimStruct macro: `ssIsMajorTimeStep`

See function in `matlabroot/rtw/c/tlc/lib/utillib.tlc`.

LibsMinorTimeStep()

Returns a string to access whether the current simulation step is a minor time step.

This function is the TLC version of the SimStruct macro `ssIsMinorTimeStep`

See function in `matlabroot/rtw/c/tlc/lib/utillib.tlc`.

LibsComplex(arg)

Returns 1 if the argument passed in is complex, 0 otherwise.

See function in `matlabroot/rtw/c/tlc/lib/utillib.tlc`.

LibsFirstInitCond(s)

`LibsFirstInitCond` 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.

This function also sets a flag that tells Real-Time Workshop if it needs to declare and maintain the `first-initialize-condition` flag.

This function is the TLC version of the SimStruct macro, `ssIsFirstInitCond`.

See function in `matlabroot/rtw/c/tlc/lib/syslib.tlc`.

LibMaxIntValue(dtype)

For a built-in integer data type, this function returns the formatted maximum value of that data type.

See function in `matlabroot/rtw/c/tlc/lib/dtypelib.tlc`.

LibMinIntValue(dtype)

For a built-in integer data type, this function returns the formatted minimum value of that data type.

See function in `matlabroot/rtw/c/tlc/lib/dtypelib.tlc`.

LibNeedAsyncCounter(sfcnBlock,callIdx)

For use by Async S-Functions with function call outputs. Asynchronous tasks can manage their own time and used this function to determine if there is a need to do.

Example

```
%if LibNeedAsyncCounter(block,0)
    %<LibSetAsyncCounter(block,0), "tickGet(">
```

Returns

Returns `TLC_TRUE` if a asynchronous counter is needed, otherwise `TLC_FALSE`.

See function in `matlabroot/rtw/c/tlc/lib/asynclib.tlc`.

LibSetAsyncCounter(sfcnBlock,callIdx,buf)

For use by Async S-Functions with function call outputs. Asynchronous tasks can manage their own time and use this function to return the counter variable that is to be maintained by the asynchronous task.

Example

```
%if LibNeedAsyncCounter(block,0)
    %<LibSetAsyncCounter(block,0), "tickGet(">
```

Returns

Returns a string for the counter variable for the asynchronous task.

See function in *matlabroot/rtw/c/tlc/lib/asynclib.tlc*.

Advanced Functions

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 this function 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 this function returns -1.

This function 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 get 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 this example could have equivalently used `(LibBlockInputSignalBufferDstPort(0) == 0)` as the Boolean condition for the `%if` statements since there is only one output port.

See function in `matlabroot/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalStorageClass(portIdx, idx)

Returns the storage class of the specified block input port signal. The storage class can be "Auto", "ExportedSignal", "ImportedExtern", or "ImportedExternPointer".

See function in `matlabroot/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalStorageTypeQualifier(portIdx, idx)

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 function in `matlabroot/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalsGlobal(portIdx)

Returns 1 if the specified block output port signal is declared in the global scope, otherwise returns 0.

If this function returns 1, then the variable holding this signal is accessible from any where in generated code. For example, this function returns 1 for signals that are test points, external or invariant.

See function in `matlabroot/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalsInBlockIO(portIdx)

Returns 1 if the specified block output port exists in the global Block I/O data structure. You may need to use this if you specify

`ssSetOutputPortReusable(S, portIdx, TRUE)` in your S-function.

See *matlabroot/toolbox/simulink/blocks/tlc_c/sfun_multiport.tlc*.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockOutputSignalsValidLValue(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, this function returns 1 if the block output port signal is in read/write memory.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.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 function in *matlabroot/rtw/c/tlc/lib/blkio.lib.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 function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockSrcSignalBlock(portIdx, idx)

Returns a reference to the block that is source of the specified block input port element. The return argument is one of the following.

[systemIdx, blockIdx]	If unique block output or block state
"ExternalInput"	If external input (root input)
"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 reused signal due to 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 function in `matlabroot/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockSrcSignalsDiscrete(portIdx, idx)

Returns 1 if the source signal corresponding to the specified block input port element is discrete, otherwise returns 0.

Note that this function also returns 0 if the driving block cannot be uniquely determined if it is a merged or reused signal (i.e., the source is a Merge block or the signal has been reused due to optimization).

See function in `matlabroot/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockSrcSignalsGlobalAndModifiable(portIdx, idx)

This function 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, this function returns 0.

See function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibBlockSrcSignalsInvariant(portIdx, idx)

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 function in *matlabroot/rtw/c/tlc/lib/blkio.lib.tlc*.

LibCreateHomogMathFcnRec(FcnName, FcnTypeId)

See function in *matlabroot/rtw/c/tlc/lib/mathlib.tlc*.

LibGetMathConstant(ConstName, ioTypeId)

Return a valid math constant expression with the proper datatype.

This function can only be called after *func.lib.tlc* is included.

See function in *matlabroot/rtw/c/tlc/lib/mathlib.tlc*.

LibMathFcnExists(RTWFcnName, RTWFcnTypeId)

Return whether or not an implementation function exists for a given generic operation (function), given the specified function prototype.

See function in *matlabroot/rtw/c/tlc/lib/mathlib.tlc*.

LibSetMathFcnRecArgExpr(FcnRec, idx, argStr)

See function in *matlabroot/rtw/c/tlc/lib/mathlib.tlc*.

TLC Error Handling

TLC Error Messages (p. A-5)

Use the `%exit` directive to generate errors from TLC files

TLC Function Library Error Messages
(p. A-30)

Messages are sufficiently self-descriptive so that they do not need additional explanation

Generating Errors from TLC-Files

To generate errors from TLC files, use the `%exit` directive, but preferably 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.
Internal 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 prepend the string `Real-Time Workshop Error` to the message you provide when reporting the error.

For an example usage 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 directories 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 may want to add an *assert* requiring that the inputs be only numerical numbers. These asserts indicate fatal coding errors in that

the user has no 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 prepend the string `Real-Time Workshop Fatal` to the message you provide and display the call stack when reporting the error.

For an example usage 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 directory `matlabroot/rtw/c/tlc`.

Using %exit

You can call `%exit` to generate fatal error messages, however, it is suggested that you use one of the previously discussed library functions. If you do use `%exit`, take care when generating an error string containing new lines (carriage returns); See “Formatting Error Messages” on page A-4.

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 RTW Fatal: error string
```

Formatting Error Messages

You should be careful when formatting error message strings. For example, suppose you create a local variable (called `message`) that contains text that has new lines.

```
%openfile message
My message text
with new lines (carriage returns)
%closefile message
```

If you then want to create another variable and prefix this message with the text “RTW Error:”, you need to use

```
%openfile errorMessage
RTW Error: %<message>
%closefile errorMessage
```

or

```
%assign errorMessage = "RTW Error:" + message
```

The statement

```
%assign errorMessage = "RTW Error: %<message>"
```

will cause a syntax error during TLC execution and your message will not be displayed. This should be avoided. Use the function `LibBlockReportError` to help prevent this type of runtime syntax error. The syntax error occurs because TLC evaluates the message, which causes new lines to appear in the assignment statement that appear as unterminated text strings (i.e., the trailing quote is missing).

After formatting your error message, use `LibBlockReportError`, a similar function, or `%exit` to report your error when it occurs.

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

This section lists and describes error messages generated by the Target Language Compiler (`tlc.mex`). Use this reference to

- Confirm that an error has been reported.
- Determine possible causes for an error.
- Determine possible ways to correct an error.

%closefile or %selectfile or %flushfile argument must be a valid open file

When using `%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 no longer supported. You must rewrite all 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 only reported 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: %s

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 a %implements directive. A %implements directive is required to ensure that the correct language and type are implemented by this block template file. See “Object-Oriented Facility for Generating Target Code” on page 5-34 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. It does this to ensure that the block-level target file implements the same language and type as specified in the %language directive.

A non-homogenous vector was passed to GENERATE_FORMATTED_VALUE

The builtin GENERATE_FORMATTED_VALUE can only process vectors which have homogenous 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

When using a repeated scope identifier from a database file, you must specify an index in order 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:

```
%assign y = block
```

In this example, the reference to block is ambiguous because multiple repeated scopes named “block” appear in the database file. Use an index to disambiguate it, as in

```
%assign y = block[0]
```

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 more 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 builtins must be records:

- 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 and 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 does not know how to cast this expression from its current type to the specified type. For example, the Target Language Compiler is not able to cast a string to a number as in

```
%assign x = "1234"  
%assign y = CAST("Number", x );
```

Bad directory (*dirname*) in O: *filename*

The -0 option was not a valid directory.

builtin* was expecting expression of type *type*, got one of type *type

A builtin was passed an expression of incorrect type.

Cannot %undef any builtin functions or variables

User is not allowed to %undef any TLC builtins or variables, 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" will contain the system specific error message.

Errors occurred — aborting

This error message is always the last error to be reported. It occurs when:

- The number of error messages exceeds the error message threshold (5 by default), or
- 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. For example,

```
%<CompiledModel.System[Sysidx].Block[BlkIdx].Name +
"Hello">
```

is illegal, whereas

```
%<CompiledModel.System[Sysidx].Block[BlkIdx].Name + \
"Hello">
```

is correct.

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 pathname, but the limits may 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 ensure that it always returns 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 integral 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

When using 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 (%d) must be less than the upper bound (%d)

When using 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 any `%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 only replace 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

When using 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. *identifier* is the name of the variable specified in the %openfile directive.

Note This might also occur if there is a syntax error in your 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, `idrange`, 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, See “Directives and Built-in Functions” on page 5-1 for information on the syntax.

The `%break` directive can only appear within a `%foreach`, `%for`, `%roll`, or `%switch` statement

The `%break` directive can only be used 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 only appear within a `%switch` statement.

The `%continue` directive can only appear within a `%foreach`, `%for`, or `%roll` statement

The `%continue` directive can only be used 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 a %if 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 only be 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 opening an output file, the name of 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 using 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 using 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 would cause 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 quotes.

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 there is no matching *end* directive. 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 has no 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 5-47.

The FEVAL() function requires the name of a function to call

FEVAL requires a function to call. This error only appears inside MATLAB.

The final argument to `%roll` must be a valid block scope

When using `%roll`, the final argument (prior to extra user-specified arguments) must be a valid block scope. See `%roll` for a complete 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 calling GENERATE or GENERATE_TYPE, the first argument must be a valid scope. See the GENERATE and GENERATE_TYPE functions 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 calling 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 calling 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 may 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 using 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 all of the rows (*actual-columns*)

When specifying a Target Language Compiler matrix, the number of columns specified did not 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 specifying a Target Language Compiler matrix, the number of rows specified did not 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 on Boolean values only.

The `operator_name` operator only works on integral arguments

The `&`, `^`, `|`, `<<`, `>>` and `%` operators only work on numbers.

The `operator_name` operator only works on numeric arguments

The arguments to the following operators both must be either Number or Real: `<`, `<=`, `>`, `>=`, `-`, `*`, `/`. This can also happen when using `+` as an unary operator. In addition, the `FORMAT` built-in function expects either a Number or Real argument.

The return value from the `RollHeader` function must be a string

When using `%roll`, the `RollHeader()` function specified in `Roller.tlc` must return a string value. See `%roll` for a complete discussion of the `%roll` construct.

The roll argument to `%roll` must be a nonempty vector of numbers or ranges

When using `%roll`, the `roll` vector cannot be empty and must contain numbers or ranges of numbers. See `%roll` for a complete discussion of the `%roll` construct.

The second value in a Range must be greater than the first value

When using 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 indexing into any 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, it returns this error.

The threshold on a %roll statement must be a single number

When using %roll, the roll threshold specified must be a single number. See %roll 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 expects two arguments: a range vector and a threshold.

There are no more free contexts. Use tlc('close', HANDLE) to free up a context

The global context table has filled up while using the TLC server mode.

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 for more information and examples on using the `GENERATE` built-in function.

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
```

The correct method is:

```
%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 a `%include` directive was not found on the path. Either locate the file and use the `-I` command line option to specify the correct directory, or move the file to a location on the current path.

Unable to open block template file *filename* from GENERATE or GENERATE_TYPE

When using GENERATE, the given filename was not found on the Target Language Compiler path. You can

- Add the file into a directory on the path.
- Use the %generatefile directive to specify an alternative filename for this block type that is on the path.
- Add the directory in which this file appears to the command line options using the -I switch.

Unable to open output file *filename*

Unable to open the specified output file; 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 calling the CAST built-in function, the type must be one of the valid Target Language Compiler types found in the Target Language Values table.

Unrecognized command line switch passed to *string*: *switch*

When querying the current state of a switch, the switch specified was not recognized.

Unrecognized directive "*directive-name*" seen

An illegal % directive was encountered. The valid directives are shown below.

%addincludepath	%filescope
%addtorecord	%for
%assert	%foreach
%assign	%function
%break	%generate

<code>%case</code>	<code>%generatefile</code>
<code>%closefile</code>	<code>%if</code>
<code>%continue</code>	<code>%implements</code>
<code>%copyrecord</code>	<code>%include</code>
<code>%createrecord</code>	<code>%language</code>
<code>%default</code>	<code>%matlab</code>
<code>%define</code>	<code>%mergerecord</code>
<code>%else</code>	<code>%openfile</code>
<code>%elseif</code>	<code>%realformat</code>
<code>%endbody</code>	<code>%return</code>
<code>%endfor</code>	<code>%roll</code>
<code>%endforeach</code>	<code>%selectfile</code>
<code>%endfunction</code>	<code>%setcommandswitch</code>
<code>%endif</code>	<code>%switch</code>
<code>%endroll</code>	<code>%trace</code>
<code>%endswitch</code>	<code>%undef</code>
<code>%endwith</code>	<code>%warning</code>
<code>%error</code>	<code>%with</code>
<code>%exit</code>	

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 produce no output.

Unterminated multiline comment.

A multiline (i.e. `!% %!`) comment has no 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

Message	Description
-r <name>	Specify the Real-Time Workshop file to read.
-v[<number>]	Specify the verbose level to be <number> (1 by default).
-I<path>	Specify a path to local include files. The TLC will search this path in the order specified.
-m[<number> a]	Specify the maximum number of errors (a is all). Default is 5.
-O<path>	Specify the path used to create output files. By default all TLC output will be created in this directory.

Message	Description
-d[a c n o]	<p>Invoke the TLC debug mode.</p> <ul style="list-style-type: none"> -da will make TLC execute any %assert directives. -dc will invoke TLC command line debugger. -dn will cause TLC to produce log files indicating which lines were and were not hit during compilation. -do will disable TLC debugging behavior.
-a<ident>=<expression>	<p>Assign a variable to a specified value. Use this option to specify parameters that can be used to change the behavior of your TLC program. This option is used by Real-Time Workshop to set options like inlining of parameters, file size limits, etc.</p>
-p<number>	<p>Print a '.' indicating progress for every <number> of TLC primitive operations executed.</p>
-lint	<p>Perform some simple performance checks and collect some runtime statistics.</p>
-x0	<p>Parse a TLC file, but not execute it.</p>

A command line problem has occurred. The error message contains a list of all 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 may cause performance hits.

Value of *specified_type* type cannot be compared

The specified type (i.e., scope) 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 can also happen when expanding a function without any arguments. If you use

```
%<Function>
```

call it with the appropriate arguments.

Values of type *Special*, *Macro Expansion*, *Function*, *File*, *Full Identifier*, and *Index* cannot be converted to MATLAB variables

The specified type cannot be 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

There are many error messages generated by the TLC function library that are not documented. These messages are sufficiently self-descriptive so that they do not need additional explanation. However, if you come across an error message that you feel needs more description, contact our technical support staff and we will update it in a future release (and give more explanation).

Using TLC with Emacs

The Emacs Editor (p. B-2)

Use the Emacs editor to edit your TLC files

The Emacs Editor

If you're editing TLC files, we recommend trying to use Emacs. You can get a copy of Emacs from <http://www.gnu.org>.

The MathWorks has created a `t1c-mode` for Emacs that gives automatic indenting and color-coded syntax highlighting of TLC files. You can obtain `t1c-mode` (and `matlab-mode`) from our Web site.

```
ftp://ftp.mathworks.com/pub/contrib/emacs_add_ons
```

See the `readme.txt` file for instructions on how to configure `t1c-mode`.

Color-coding syntax in Emacs makes TLC code is much more readable.

Getting Started

To get started using Emacs:

Ctrl+x Ctrl+f `file.t1c` <return> Loads a file into an Emacs buffer for editing.

Ctrl+x Ctrl+s Saves the file in the current buffer.

Ctrl+x Ctrl+c Exits Emacs.

Ctrl stands for control key. For example, to load a file into Emacs, hold down the control key and type `x`, followed by `f` with the control key still pressed, then release the control key and type the name of a file followed by return. A tutorial is available from the Emacs Help menu.

Creating a TAGS File

If you are familiar with Emacs TAGS, you can create a TAGS file for TLC files by invoking

```
etags --regex='/[ \t]*\%function[ \t]+.+/' --language=none *.t1c
```

in the UNIX directory where your `.t1c` files are located. The `etags` command is located the `emacs_root/bin` directory. Users of Windows NT must type

```
etags "--regex=[ \t]*\%function[ \t]+.+/" --language=none *.t1c
```

in a DOS command window.

Symbols

- ! 5-23
- 5-23
- 5-24
- != 5-25
- % 5-2, 5-21, 5-23
- & 5-25
- && 5-25
- () 5-23
- * 5-23
- + 5-23, 5-24
- , 5-26
- . 5-23
- ... 5-18
- .c file 1-5
- .h file 1-5
- .log 6-9
- .rtw file 1-5
- / 5-23
- :: 5-22, 5-52
- < 5-25
- << 5-24
- <= 5-25
- == 5-25
- > 5-24
- >= 5-25
- >> 5-24
- ? : 5-26
- \ 5-18
- ^ 5-25
- _prm.h file 1-5
- _reg.h file 1-5
- | 5-25
- || 5-25
- ~ 5-23

A

- %addincludepath 5-37
- array index 5-22
- %assert 5-38
- assert
 - adding A-2
- %assign 5-51, 7-24
 - defining parameters 3-19

B

- block
 - customizing Simulink 5-34
- block function 7-29
 - InitializeConditions 7-34
 - Start 7-34
- block target file 1-4, 7-29
 - function in 7-25
 - mapping 3-25
 - writing 7-30
- BlockInstanceSetup 7-30
- block-scoped variable 5-58
- BlockTypeSetup 7-31
- %body 5-30
- Boolean 5-19
- %break 5-29, 5-30
- %continue 5-29
- buffer
 - close 5-37
 - writing 5-36
- built-in functions 5-39
 - CAST 5-40
 - EXISTS 5-40
 - FEVAL 5-40
 - FIELDNAMES 5-41
 - FILE_EXISTS 5-40

FORMAT 5-41
GENERATE 5-41
GENERATE_FILENAME 5-41
GENERATE_FORMATTED_VALUE 5-41
GENERATE_FUNCTION_EXISTS 5-42
GENERATE_TYPE 5-42
GENERATE_TYPE_FUNCTION_EXISTS 5-42
GET_COMMAND_SWITCH 5-42
GETFIELD 5-41
IDNUM 5-42
IMAG 5-42
INT16MAX 5-42
INT16MIN 5-42
INT32MAX 5-42
INT32MIN 5-43
INT8MAX 5-42
INT8MIN 5-42
INTMAX 5-43
INTMIN 5-43
ISALIAS 5-43
ISEMPTY 5-43
ISEQUAL 5-43
ISFIELD 5-43
ISFINITE 5-43
ISINF 5-43
ISNAN 5-43
NULL_FILE 5-44
NUMTLCFILES 5-44
OUTPUT_LINES 5-44
REAL 5-44
REMOVEFIELD 5-44
ROLL_ITERATIONS 5-44
SETFIELD 5-44
SIZE 5-45
SPRINTF 5-45
STDOUT 5-45
STRING 5-45

STRINGOF 5-45
SYSNAME 5-46
TLC_FALSE 5-46
TLC_TIME 5-46
TLC_TRUE 5-46
TLC_VERSION 5-46
TLCFILES 5-46
TYPE 5-46
UINT16MAX 5-47
UINT32MAX 5-47
UINT8MAX 5-47
UINTMAX 5-47
WHITE_SPACE 5-47
WILL_ROLL 5-47

C

C MEX S-function 1-4
%case 5-29
CAST 5-40
%closefile 5-36
code
 intermediate 3-18
code coverage 6-9
code generation 1-9
coding conventions 7-24
comment
 target language 5-17
CompiledModel 4-3
Compiler
 Target Language (TLC) 1-2
Complex 5-19
Complex32 5-19
conditional
 inclusion 5-28
 operator 5-21
constant

- integer 5-21
- string 5-21
- continuation
 - line 5-18
- %continue 5-30
- customizing
 - code generation 3-18
 - Simulink block 5-34

D

- debug
 - message 5-38
- debugger 6-2, 6-5
- debugger commands
 - viewing 6-5
- debugging tips 6-2
- %default 5-29
- Derivatives 7-36
- directive 3-19, 5-2
 - object-oriented 5-34
 - splitting 5-18
- directives
 - %% 5-2
 - %<expr> 5-3
 - %addincludepath 5-10
 - %addtorecord 5-8
 - %assert 5-5
 - %assign 5-6
 - %break 5-4
 - %case 5-4
 - %closefile 5-16
 - %copyrecord 5-8
 - %createrecord 5-7
 - %default 5-4
 - %else 5-4
 - %elseif 5-4
 - %endforeach 5-15
 - %endfunction 5-13
 - %endif 5-4
 - %endroll 5-11
 - %endswitch 5-4
 - %endwith 5-5
 - %error 5-6
 - %exit 5-6
 - %filescope 5-10
 - %for 5-16
 - %foreach 5-15
 - %function 5-13
 - %generatefile 5-9
 - %if 5-4
 - %implements 5-9
 - %include 5-10
 - %language 5-9
 - %matlab 5-2
 - %mergerecord 5-8
 - %openfile 5-16
 - %realformat 5-8
 - %return 5-13
 - %roll 5-11
 - %selectfile 5-16
 - %setcommandswitch 5-5
 - %switch 5-4
 - %trace 5-6
 - %warning 5-6
 - %with 5-5
 - /% text %/ 5-2
- Disable 7-32
- dynamic scoping 5-56

E

- %else 5-28
- %elseif 5-28

- Enable 7-32
- %endbody 5-30
- %endfor 5-30
- %endforeach 5-29
- %endfunction 5-66
- %endif 5-28
- %endswitch 5-29
- %endwith 5-58
- %error 5-38
- error
 - formatting messages A-4
 - internal A-2
 - usage A-2
- error message 5-38
 - Target Language Compiler A-5
- EXISTS 5-40
- %exit 5-39
- expressions 5-21
 - operators in 5-21
 - precedence 5-21
- F**
- FEVAL 5-40
- FIELDNAMES 5-41
- File 5-19
- file
 - .c 1-5
 - .h 1-5
 - .rtw 1-5
 - _prm.h 1-5
 - _reg.h 1-5
 - appending 5-37
 - block target 1-4, 3-22
 - close 5-37
 - inline 5-37
 - model description. *See* model.rtw
 - model-wide target 3-19
 - system target 3-23
 - target 1-4, 3-18
 - target language 3-25
 - used to customize code 3-18
 - writing 5-36
- FILE_EXISTS 5-40
- %for 5-30
- %foreach 5-29
- FORMAT 5-41
- formatting 5-27
- Function 5-19
- %function 5-66
- function
 - C MEX S-function 1-4
 - call 5-22
 - GENERATE 5-35
 - GENERATE_TYPE 5-35
 - library 7-27
 - output 5-67
 - target language 5-66
 - Target Language Compiler 5-39–5-47
- functions
 - obsolete 8-2
- G**
- Gaussian 5-19
- Gaussian, Unsigned 5-20
- GENERATE 5-35, 5-41
- GENERATE_FILENAME 5-41
- GENERATE_FORMATTED_VALUE 5-41
- GENERATE_FUNCTION_EXISTS 5-42
- GENERATE_TYPE 5-35, 5-42
- GENERATE_TYPE_FUNCTION_EXISTS 5-42
- %generatefile 5-34
- GET_COMMAND_SWITCH 5-42

GETFIELD 5-41

I

identifier 7-24
 changing 5-51
 defining 5-51
IDNUM 5-42
%if %endif 5-28
IMAG 5-42
%implements 5-34
%include 5-37
inclusion
 conditional 5-28
 multiple 5-29
index 5-22
Initialize 7-34
InitializeConditions 7-34
inlining S-function 7-5
 advantages 1-13
input file control 5-37
INT16MAX 5-42
INT16MIN 5-42
INT32MAX 5-42
INT32MIN 5-43
INT8MAX 5-42
INT8MIN 5-42
integer constant 5-21
intermediate code 3-18
INTMAX 5-43
INTMIN 5-43
ISALIAS 5-43
ISEMPTY 5-43
ISEQUAL 5-43
ISFIELD 5-43
ISFINITE 5-43
ISINF 5-43

ISNAN 5-43

L

%language 5-34
lcv, definition 8-4
library functions
 LibBlockContinuousState 8-31
 LibBlockDiscreteState 8-33
 LibBlockDWork 8-31
 LibBlockDWorkAddr 8-32
 LibBlockDWorkDataTypeId 8-32
 LibBlockDWorkDataTypeName 8-32
 LibBlockDWorkIsComplex 8-32
 LibBlockDWorkName 8-32
 LibBlockDWorkStorageClass 8-32
 LibBlockDWorkUsedAsDiscreteState 8-33
 LibBlockDWorkWidth 8-33
 LibBlockInputSignal 8-9
 LibBlockInputSignalAddr 8-16
 LibBlockInputSignalBufferDstPort 8-78
 LibBlockInputSignalConnected 8-17
 LibBlockInputSignalDataTypeId 8-17
 LibBlockInputSignalDataTypeName 8-17
 LibBlockInputSignalDimensions 8-18
 LibBlockInputSignalIsComplex 8-18
 LibBlockInputSignalIsFrameData 8-18
 LibBlockInputSignalLocalSampleTimeIndex
 8-18
 LibBlockInputSignalNumDimensions 8-18
 LibBlockInputSignalOffsetTime 8-18
 LibBlockInputSignalSampleTime 8-19
 LibBlockInputSignalSampleTimeIndex 8-19
 LibBlockInputSignalStorageClass 8-79
 LibBlockInputSignalStorageTypeQualifier
 8-79
 LibBlockInputSignalWidth 8-19

LibBlockIWork 8-33
LibBlockMatrixParameter 8-26
LibBlockMatrixParameterAddr 8-26
LibBlockMode 8-33
LibBlockNonSampledZC 8-33
LibBlockOutputSignal 8-21
LibBlockOutputSignalAddr 8-21
LibBlockOutputSignalBeingmerged 8-22
LibBlockOutputSignalConnected 8-22
LibBlockOutputSignalDataTypeId 8-23
LibBlockOutputSignalDataTypeName 8-23
LibBlockOutputSignalDimensions 8-23
LibBlockOutputSignalIsComplex 8-23
LibBlockOutputSignalIsFrameData 8-23
LibBlockOutputSignalIsGlobal 8-79
LibBlockOutputSignalIsInBlockIO 8-80
LibBlockOutputSignalIsValidLValue 8-80
LibBlockOutputSignalLocalSampleTimeIndex 8-24
LibBlockOutputSignalNumDimensions 8-24
LibBlockOutputSignalOffsetTime 8-24
LibBlockOutputSignalSampleTime 8-24
LibBlockOutputSignalSampleTimeIndex 8-24
LibBlockOutputSignalStorageClass 8-80
LibBlockOutputSignalStorageTypeQualifier 8-80
LibBlockOutputSignalWidth 8-24
LibBlockParameter 8-27
LibBlockParameterAddr 8-28
LibBlockParameterBaseAddr 8-29
LibBlockParameterDataTypeId 8-29
LibBlockParameterDataTypeName 8-29
LibBlockParameterDimensions 8-29
LibBlockParameterIsComplex 8-30
LibBlockParameterSize 8-30
LibBlockPWork 8-34
LibBlockReportError 8-35
LibBlockReportFatalError 8-35
LibBlockReportWarning 8-35
LibBlockRWork 8-34
LibBlockSampleTime 8-59
LibBlockSrcSignalBlock 8-80
LibBlockSrcSignalIsDiscrete 8-81
LibBlockSrcSignalIsGlobalAndModifiable 8-81
LibBlockSrcSignalIsInvariant 8-82
LibCacheDefine 8-38
LibCacheExtern 8-38
LibCacheFunctionPrototype 8-39
LibCacheIncludes 8-39
LibCacheTypedefs 8-39
LibCallFCSS 8-67
LibGetBlockPath 8-36
LibGetDataTypeIdAliasedToFromId 8-74
LibGetDataTypeIdComplexNameFromId 8-74
LibGetDataTypeIdEnumFromId 8-74
LibGetDataTypeIdNameFromId 8-74
LibGetDataTypeIdResolvesToFromId 8-74
LibGetFormattedBlockPath 8-36
LibGetGlobalTIDFromLocalSFcnTID 8-61
LibGetNumSFcnSampleTimes 8-62
LibGetSFcnTIDType 8-63
LibGetTaskTimeFromTID 8-63
LibIsComplex 8-75
LibIsContinuous 8-63
LibIsDiscrete 8-64
LibIsFirstInitCond 8-75
LibIsSFcnSampleHit 8-64
LibIsSFcnSingleRate 8-65
LibIsSFcnSpecialSampleHit 8-65
LibMaxIntValue 8-76
LibMinIntValue 8-76
LibSetVarNextHitTime 8-66

M

- macro
 - expansion 5-22
- makefile
 - template 1-4
- Matrix 5-19
- mdlDerivatives (S-function) 7-5
- mdlInitializeConditions 7-5
- mdlInitializeSampleTimes 7-5
- mdlInitializeSizes 7-5
- mdlOutputs (S-function) 7-5
- mdlRTW method
 - registering parameters with 7-6
- MdlStart
 - InitializeConditions 7-34
- MdlTerminate
 - Terminate 7-36
- mdlTerminate (S-function) 7-5
- mdlUpdate (S-function) 7-5
- model description file. *See* model.rtw
- model.rtw file 1-3, 5-39
- model-wide target file 3-19
- modifier
 - Output 5-67
 - void 5-67
- multiple inclusion 5-29

N

- negation operator 5-23
- nested function
 - scope within 5-68
- new features
 - version 4.0 1-18
 - version 4.1 1-17
 - version 5.0 1-16
- NULL_FILE 5-44

- Number 5-19
- NUMTLCFILES 5-44

O

- object-oriented directive 5-34
- obsolete functions 8-2
- %openfile 5-36
- operations
 - precedence 5-23
- operator 5-21
 - :: 7-25
 - conditional 5-21
 - negation 5-23
- output file control 5-36
- Output modifier 5-67
- OUTPUT_LINES 5-44
- Outputs 7-34

P

- parameter
 - defining 3-19
 - value pair 4-2
- parameter settings
 - in s-functions 7-6
- paramIdx 8-6
- path
 - specifying absolute 5-38
 - specifying relative 5-38
- portIdx, definition 8-4
- precedence
 - expressions 5-21
 - operations 5-23
- profiler 6-13
 - using 6-13
- program 3-25

R

- Range 5-20
- REAL 5-44
- Real 5-20
- Real32 5-20
- %realformat 5-27
- Real-Time Workshop 1-2
- record 3-14, 4-2
- REMOVEFIELD 5-44
- resolving variables 5-56
- %return 5-66, 5-70
- %roll
 - common arguments to 8-4
 - directive 5-11
 - syntax of 5-31
- ROLL_ITERATIONS 5-44
- rt 7-26
- rt_ 7-26
- RTW
 - identifier 7-24

S

- Scope 5-20
- scope 5-56
 - accessing values in 4-3
 - closing 5-70
 - dynamic 5-56
 - function 5-22
 - within function 5-68
- search path 5-73
 - adding to 5-38
 - overriding 5-73
 - sequence 5-38
 - specifying absolute 5-38
 - specifying relative 5-38
- %selectfile 5-36

- SETFIELD 5-44

S-function

- advantage of inlining 1-13
- C MEX 1-4
- inlining 7-5
- user-defined 7-36

- sigIdx 8-5

Simulink

- and Real-Time Workshop 1-2
- generating code 1-5

Simulink data objects

- and ObjectProperties records 4-6

- SIZE 5-45

- Special 5-20

- SPRINTF 5-45

- Start 7-33

- stateIdx 8-6

- STDOUT 5-45

- STRING 5-45

- String 5-20

- string constant 5-21

- string variables

- white space in 3-11

- STRINGOF 5-45

- substitution

- textual 5-21

- Subsystem 5-20

- %switch 5-29

- syntax 5-2

- SYS_NAME 5-46

- system target file 3-23

T

- target file 1-4, 3-18, 3-22

- and customizing code 3-18

- block 3-25, 7-29

- model-wide 3-19
 - naming 5-73
 - system 3-23
 - target language 3-13
 - comment 5-17
 - directive 3-19, 5-2
 - expression 5-21–5-26
 - file 5-2
 - formatting 5-27
 - function 5-66
 - line continuation 5-18
 - program 3-25
 - syntax 5-2
 - value 5-19–5-20
 - Target Language Compiler
 - a parameter 3-10
 - command line arguments 5-71
 - configuring 3-10
 - directives 5-2–5-17
 - error messages A-5
 - function library 7-27
 - introducing 1-2
 - switches 5-71
 - uses of 1-7
 - variables 7-26
 - template makefile 1-4
 - Terminate 7-36
 - textual substitution 5-21
 - TLC code
 - debugging tips 6-2
 - TLC coverage option 6-9
 - TLC debugger 6-2
 - TLC debugger commands 6-5
 - TLC profiler 6-13
 - TLC program 3-25
 - TLC_FALSE 5-46
 - TLC_TIME 5-46
 - TLC_TRUE 5-46
 - TLC_VERSION 5-46
 - TLCFILES 5-46
 - %trace 5-39
 - tracing 5-39
 - tunable parameters
 - in s-functions 7-6
 - TYPE 5-46
- ## U
- ucv, definition 8-4
 - UINT16MAX 5-47
 - UINT32MAX 5-47
 - UINT8MAX 5-47
 - UINTMAX 5-47
 - Unsigned 5-20
 - Unsigned Gaussian 5-20
 - Update 7-36
- ## V
- values 5-19
 - variables
 - block-scoped 5-58
 - global 7-25
 - local 7-25
 - Vector 5-20
 - void modifier 5-67
- ## W
- %warning 5-39
 - warning message 5-38
 - WHITE_SPACE 5-47
 - WILL_ROLL 5-47
 - %with 5-58

Z

zero-crossing

reset code 7-36