

Target Language CompilerTM

For Use with Real-Time Workshop[®]

Modeling
└─

Simulation
└─

Implementation
└─



Reference Guide
Version 4

How to Contact The MathWorks:



508-647-7000

Phone



508-647-7001

Fax



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

Mail



<http://www.mathworks.com>
<ftp.mathworks.com>
<comp.soft-sys.matlab>

Web
Anonymous FTP server
Newsgroup



support@mathworks.com
suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
subscribe@mathworks.com
service@mathworks.com
info@mathworks.com

Technical support
Product enhancement suggestions
Bug reports
Documentation error reports
Subscribing user registration
Order status, license renewals, passcodes
Sales, pricing, and general information

Target Language Compiler Reference Guide

© COPYRIGHT 1997 - 2000 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 or for the federal government of the United States. By accepting delivery of the Program, the government hereby agrees that this software qualifies as "commercial" computer software within the meaning of FAR Part 12.212, DFARS Part 227.7202-1, DFARS Part 227.7202-3, DFARS Part 252.227-7013, and DFARS Part 252.227-7014. The terms and conditions of The MathWorks, Inc. Software License Agreement shall pertain to the government's use and disclosure of the Program and Documentation, and shall supersede any conflicting contractual terms or conditions. If this license fails to meet the government's minimum needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to MathWorks.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and Target Language Compiler 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 for Target Language Compiler 1.0
September 2000 Updated for Version 4/Release 12 (online only)

Introducing the Target Language Compiler

1

Overview of the TLC Process	1-3
Overview of the Code Generation Process	1-5
Capabilities	1-7
Customizing Output	1-7
Inlining S-Functions	1-7
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
Process Specifics For The Real-Time Workshop Ada Coder ..	1-12
Advantages of Inlining S-Functions	1-14
Motivations	1-14
Inlining Process	1-15
Search Algorithm for Locating Target Files	1-16
Availability for Inlining and Noninlining	1-16
Target Language Compiler 4.0 New Features	1-17
Compatibility Issues	1-18
Where to Go from Here	1-21
Related Manuals	1-21

Getting Started

2

Code Architecture	2-2
Model.rtw and TLC Overview	2-4
The TLC 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 TLC to Generate Code	3-7
Code Generation Concepts	3-8
Output Streams	3-8
Variable Types	3-9
Records	3-9
Record Aliases	3-11
TLC Files	3-14
Introducing Target Files	3-14
System Target Files	3-16
Block Target Files	3-17
Block Target File Mapping	3-17
Target Files	3-17
Writing Target Language Files: A Tutorial	3-23
Matrix Parameters in Real-Time Workshop	3-23
Configuring TLC	3-26

4

Overview of model.rtw File	4-2
Using Scopes in the model.rtw File	4-3
Using Library Functions to Access model.rtw Contents ...	4-6
Caution Against Directly Accessing Record Fields	4-6
Exception to Using the Library Functions	4-7

Directives and Built-in Functions

5

Compiler Directives	5-2
Syntax	5-2
Comments	5-15
Line Continuation	5-16
Target Language Values	5-17
Target Language Expressions	5-19
Formatting	5-25
Conditional Inclusion	5-25
Multiple Inclusion	5-27
Object-Oriented Facility for Generating Target Code	5-32
Output File Control	5-34
Input File Control	5-35
Asserts, Errors, Warnings, and Debug Messages	5-36
Built-In Functions and Values	5-37
TLC Reserved Constants	5-50
Identifier Definition	5-51
Scoping	5-55
Target Language Functions	5-59
Command Line Arguments	5-65
Filenames and Search Paths	5-66

About the TLC Debugger	6-2
Tips for Debugging TLC Code	6-2
Using the TLC Debugger	6-3
Tutorial	6-3
TLC Debugger Commands	6-6
Example TLC Debugging Session	6-9
TLC Coverage	6-11
Using the TLC Coverage Option	6-11
TLC Profiler	6-14
Using the Profiler	6-14

Inlining S-Functions

Writing Block Target Files to Inline S-Functions	7-2
Fully Inlined S-Functions	7-2
Function-Based or Wrappered Code Generation	7-2
Inlining C MEX S-Functions	7-4
S-Function Parameters	7-5
A Complete Example	7-6
Inlining M-File S-Functions	7-17

Inlining Fortran (FMEX) S-Functions	7-19
Inlining Ada-MEX S-Functions	7-23
TLC Coding Conventions	7-26
Block Target File Methods	7-31
Block Target File Mapping	7-31
Block Functions	7-31
Loop Rolling	7-41
Error Reporting	7-43

TLC Tutorial

8	<table> <tr> <td>Syntax Highlighting with Emacs</td> <td>8-2</td> </tr> <tr> <td>Basic Inlined S-Function Written in TLC</td> <td>8-3</td> </tr> <tr> <td> Basic Code Generation</td> <td>8-3</td> </tr> <tr> <td> Creating an Inlined S-Function</td> <td>8-4</td> </tr> <tr> <td>Introduction to TLC Token Expansion</td> <td>8-6</td> </tr> <tr> <td>Building a Model Using the TLC Debugger</td> <td>8-8</td> </tr> <tr> <td> Preparing the Example</td> <td>8-8</td> </tr> <tr> <td> Performing the Tasks</td> <td>8-8</td> </tr> <tr> <td>Explore Variable Names and Loop Rolling</td> <td>8-10</td> </tr> <tr> <td> Preparing the Example</td> <td>8-10</td> </tr> <tr> <td> Performing the Tasks</td> <td>8-10</td> </tr> <tr> <td>Code Coverage for Debugging TLC Files</td> <td>8-13</td> </tr> <tr> <td> Performing the Tasks</td> <td>8-13</td> </tr> <tr> <td>Using a Wrapper S-Function Inlined with TLC</td> <td>8-16</td> </tr> </table>	Syntax Highlighting with Emacs	8-2	Basic Inlined S-Function Written in TLC	8-3	Basic Code Generation	8-3	Creating an Inlined S-Function	8-4	Introduction to TLC Token Expansion	8-6	Building a Model Using the TLC Debugger	8-8	Preparing the Example	8-8	Performing the Tasks	8-8	Explore Variable Names and Loop Rolling	8-10	Preparing the Example	8-10	Performing the Tasks	8-10	Code Coverage for Debugging TLC Files	8-13	Performing the Tasks	8-13	Using a Wrapper S-Function Inlined with TLC	8-16
Syntax Highlighting with Emacs	8-2																												
Basic Inlined S-Function Written in TLC	8-3																												
Basic Code Generation	8-3																												
Creating an Inlined S-Function	8-4																												
Introduction to TLC Token Expansion	8-6																												
Building a Model Using the TLC Debugger	8-8																												
Preparing the Example	8-8																												
Performing the Tasks	8-8																												
Explore Variable Names and Loop Rolling	8-10																												
Preparing the Example	8-10																												
Performing the Tasks	8-10																												
Code Coverage for Debugging TLC Files	8-13																												
Performing the Tasks	8-13																												
Using a Wrapper S-Function Inlined with TLC	8-16																												

When to Consider?	8-16
Exercise	8-16
The TLC Wrapper	8-17
Solution	8-18
Inlined S-Function for Dual Port RAM	8-20
Exercise	8-20
Further Hints	8-22
Solution	8-23
“Hello World” Example with model.rtw File	8-24
Exercise	8-24
Generating Auxiliary Files for Batch FTP	8-25
Exercise	8-25
Generating Code for Models with States	8-26
Derivatives Function	8-29
Simulink External Mode and GRT	8-35
Running External Mode in Simulink	8-35
Advanced External Mode Exercise	8-38
Loop Rolling Through a TLC File	8-40
Arguments for %roll	8-40
Input Signals, Output Signals, and Parameters	8-41

TLC Function Library Reference

9

Obsolete Functions	9-3
Target Language Compiler Functions	9-5
Common Function Arguments	9-5

Input Signal Functions	9-10
Output Signal Functions	9-15
Parameter Functions	9-18
Block State and Work Vector Functions	9-22
Block Path and Error Reporting Functions	9-25
Code Configuration Functions	9-27
Sample Time Functions	9-30
Other Useful Functions	9-36
Advanced Functions	9-39

A

model.rtw

model.rtw File Contents	A-2
Understanding the model.rtw File	A-2
General model.rtw Concepts	A-5
model.rtw Changes Between Real-Time Workshop	
3.0 and 4.0	A-6
model.rtw Differences	A-6
General Information and Solver Specification	A-11
RTWGenSettings Record	A-13
Data Logging Information	A-14
Data Structure Sizes	A-16

Sample Time Information	A-18
Data Type Information	A-20
Block Type Counts	A-21
Model Hierarchy	A-22
External Inputs and Outputs	A-25
Data Store Information	A-27
Block I/O Information	A-28
Data Type Work (DWork) Information	A-33
State Mapping Information	A-35
Block Record Defaults	A-36
Parameter Record Defaults	A-37
Data and Control Port Defaults	A-38
Model Parameters Record	A-40
System Record	A-43
Stateflow Record	A-56
Model Checksums	A-57
Block Specific Records	A-58
Linear Block Specific Records	A-77

TLC Error Handling

B

Generating Errors from TLC-Files	B-2
Usage Errors	B-2
Fatal (Internal) TLC Coding Errors	B-2
Formatting Error Messages	B-3
 TLC Error Messages	B-5
 TLC Function Library Error Messages	B-32

Using TLC with Emacs

C

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

Introducing the Target Language Compiler

Overview of the TLC Process	1-3
Overview of the Code Generation Process	1-5
Capabilities	1-7
Customizing Output	1-7
Inlining S-Functions	1-7
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
Process Specifics For The Real-Time Workshop Ada Coder	1-12
Advantages of Inlining S-Functions	1-14
Motivations	1-14
Inlining Process	1-15
Search Algorithm for Locating Target Files	1-16
Availability for Inlining and Noninlining	1-16
Target Language Compiler 4.0 New Features	1-17
Compatibility Issues	1-17
Where to Go from Here	1-21
Related Manuals	1-21

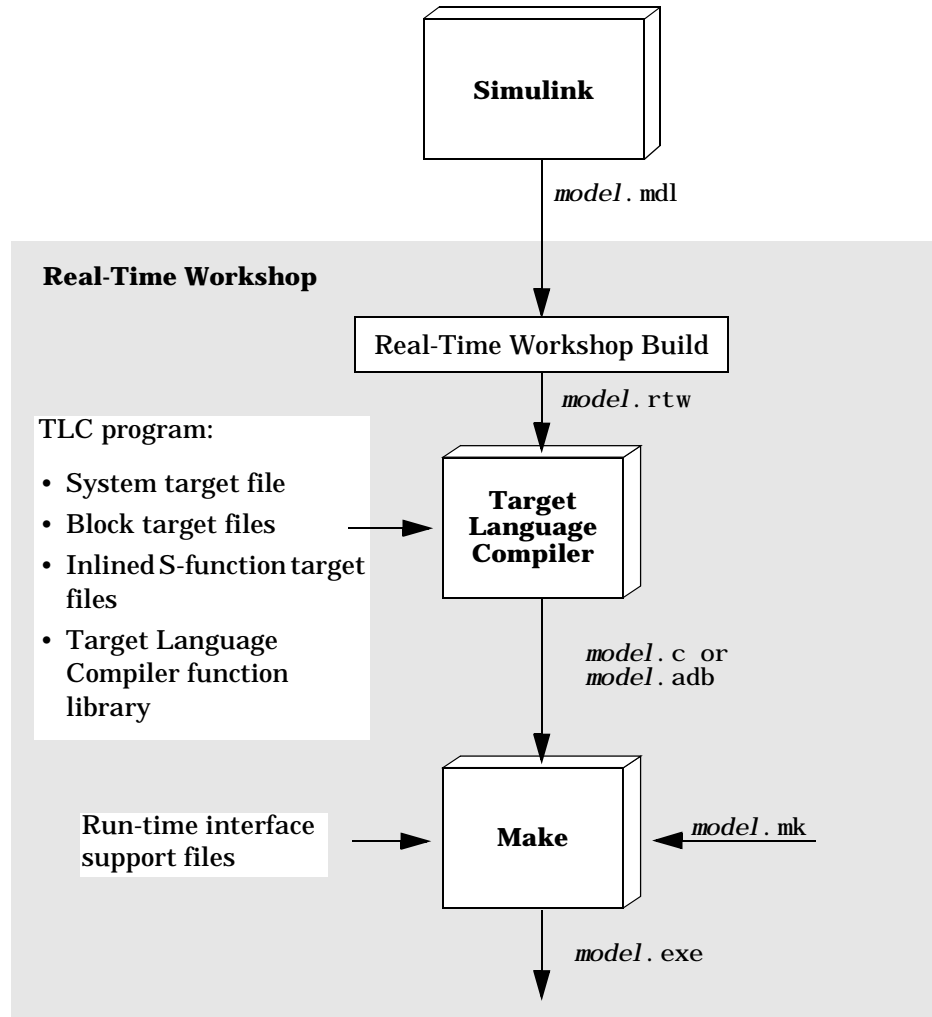
The Target Language Compiler™ tool is an integral part of the Real-Time Workshop®. It enables you to customize the C or Ada code generated from any Simulink® model and generate optimal, inlined code for your own Simulink blocks. Through customization, you can produce platform-specific code, or you can 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, in some cases, for users who want to inline S-functions 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 the *Real-Time Workshop User's Guide* in the “Code Generation and the Build Process” chapter.

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

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.



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 or Ada 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. 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. TLC works like a text processor, using the target files and the *model.rtw* file to generate ANSI C or Ada 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 User's Guide*.

For those familiar with HTML, Perl, and MATLAB®, you will find that the Target Language Compiler borrows ideas from each of them. It has the mark-up-like notion of HTML, and the power and flexibility of Perl and other scripting languages. It has the data handling power of MATLAB. The Target Language Compiler is designed for one purpose — to convert the model description file, *model.rtw*, (or similar files) into target specific code or text.

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, thus improving performance by eliminating function calls to the S-function itself and the memory overhead associated with 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 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 Chapter 7, "Inlining S-Functions." You can also write target files for M-files or Fortran S-functions.

Overview of the Code Generation Process

Figure 1-1 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.

Note In the case of the S-function target, a stub-included version of the final *model_sf.c* file is written to the current directory for use when the S-function is used in a model that is in turn used for code generation.

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.

Files placed in the build directory include:

- 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

The process to create Ada source files is similar to that for creating C source files.

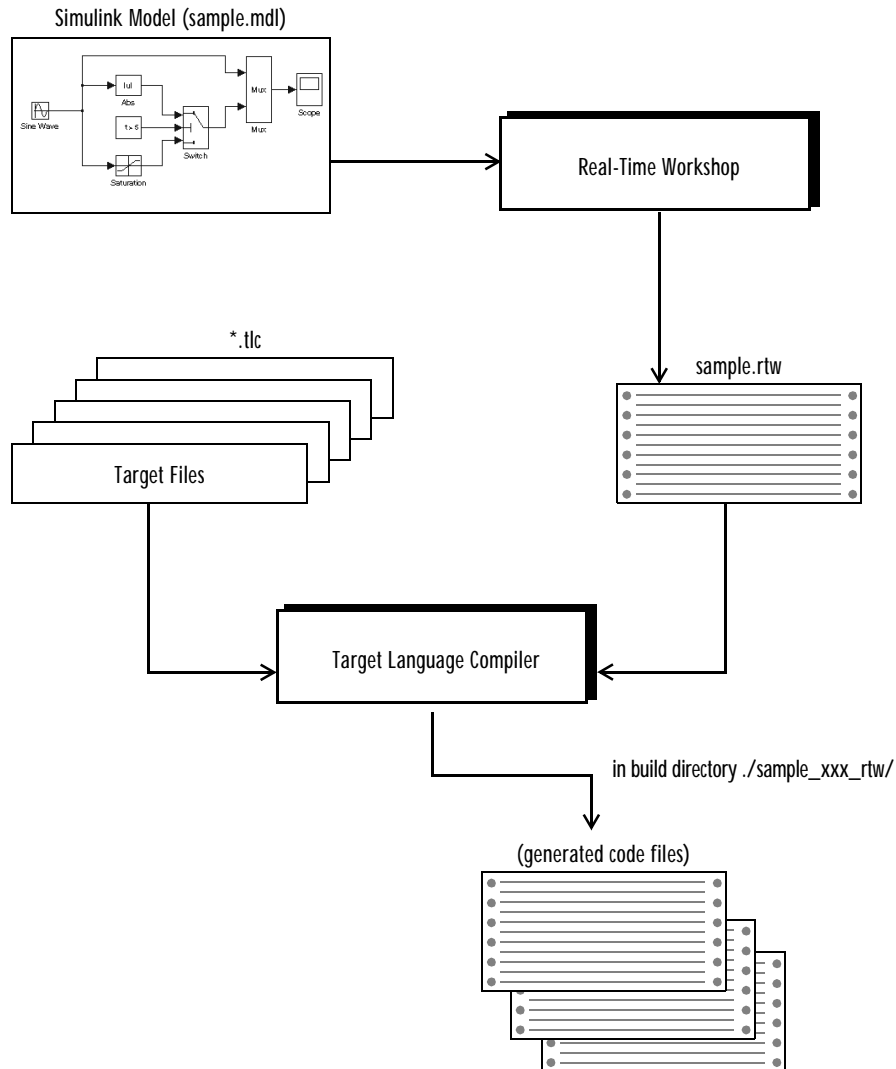


Figure 1-1: The Target Language Compiler Process

Capabilities

If you simply need to produce ANSI C or Ada code from a Simulink model, you do not need to use the Target Language Compiler. If you need to customize the output of Real-Time Workshop, the Target Language Compiler is the mechanism that you would use. Use the Target Language Compiler if you need to:

- Change the way code is generated for a particular Simulink block
- Inline S-functions in your model
- Modify the way code is generated in a global sense
- Perform a large scale customization of the generated code, for example, if you need to output the code in a language other than C

Customizing Output

To produce customized output using the Target Language Compiler, you need to understand the structure of the *model.rtw* file and how to modify target files to produce the desired output. The “Directives and Built-In Functions” chapter 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. The “Writing Target Language Files: A Tutorial” section in the “Code Generation Architecture” chapter explains the details of writing target files.

Inlining S-Functions

The Target Language Compiler provides a great deal of freedom for altering or enhancing the generated code. One of the most important features of the Target Language Compiler is that it lets you inline S-functions, since S-functions let you enhance Simulink by adding your own algorithms, device drivers, and so on to a Simulink model.

To create an S-function, you write C code following a well-defined API. When generating code, a noninlined S-function is called using this same API. There is a fair amount of overhead in having a common API in that a large data structure called the SimStruct is maintained for each instance of an S-function block in your model. In addition, there is extra overhead for calling the methods (functions) within your S-function. You can eliminate this overhead by inlining the S-function using TLC. This is done by creating a TLC file named

sfunction_name.tlc. Inlining an S-function improves the efficiency and reduces memory usage of the generated code.

Technically, you can use the Target Language Compiler to convert the *model.rtw* file into any form of output by replacing all of the TLC files. You can also replace some or all of the shipping system-wide and built-in block TLC files. This is supported but not recommended. If you choose to perform such operations, you may need to update your TLC files with each release of the Real-Time Workshop. The MathWorks continues to improve the code generator by adding features and making it more efficient. With each release, The MathWorks may alter the contents of the *model.rtw* file. We try to make it backwards compatible, but cannot guarantee this. However, inlined TLC files are generally backwards compatible, providing they use the documented Lib* TLC functions.

Code Generation Process

Real-Time Workshop invokes TLC after a Simulink model is compiled into an intermediate form that is suitable for generating code. In addition to TLC's library of functions, the two main classes of target files are:

- System target files
- Block target files

System target files are used to specify the overall structure of the generated code. 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, Ada 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. Ada and Fortran S-functions must be wrapper 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 has to decide either to generate a call to the S-function or inline it.

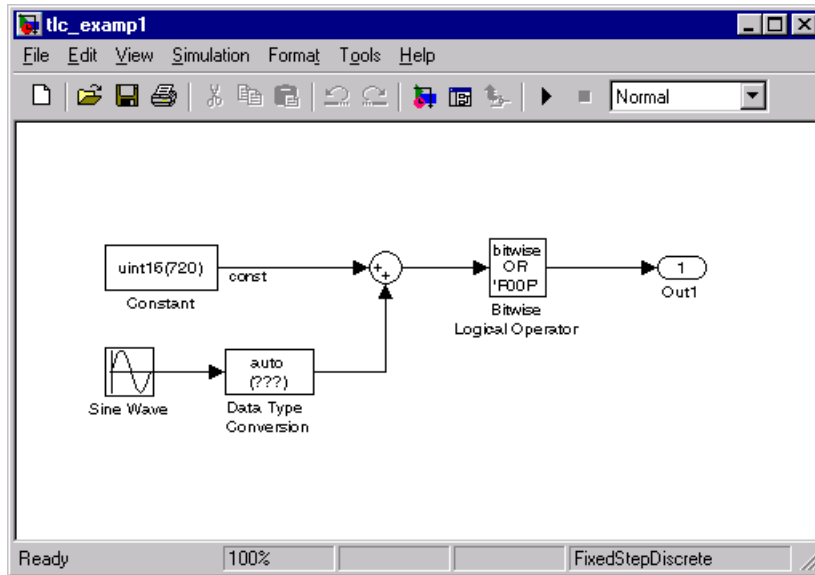
Ada, Fortran, and M-file S-functions must be inlined. This inlining can either be in the form of a full block target file or a “one-liner” block target file that references a “substitute” C MEX S-function source file.

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_biotop.c` has its compiled form in `toolbox/simulink/blocks/sfun_biotop.dll` (`.mex*` for UNIX) and its C target file is located in `toolbox/simulink/blocks/tlc_c/sfun_biotop.tlc`. This example S-function also has an Ada target file, which is located in `toolbox/simulink/blocks/tlc_ada/sfun_biotop.tlc`, if you have the Real-Time Workshop Ada Coder.

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 model's 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 codebase, 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 mdl Outputs 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 mdl Outputs 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 the 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.

Process Specifics For The Real-Time Workshop Ada Coder

The code generated by the Ada Coder uses the same strategy and algorithms as the equivalent C target file. Minor language-specific differences will exist, such as when C code invokes a call to the C standard library `memcpy()` function to copy an array of data, the generated Ada code will generate an assignment loop.

When searching for inlined S-Function block target files, the Ada Coder instructs the Target Language Compiler to look in `tlc_ada/` directories instead of `tlc_c/` directories. Target files for Ada have the `%implements ... "Ada"` directive in them instead of the `%implements ... "C"` directive.

The filenames generated by the Ada Coder follow the recommended extension naming practice used by the GNU Ada Translator (GNAT) environment, which are the familiar `.adb` (body), `.ads` (specification) file extensions.

The generated code is formatted for capitalization, comment style, and indentation to conform to the conventions in the book *Ada 95 Quality and Style: Guidelines for Professional Programmers*, part no. SPC-94093-CMC, version

01.00.10 from the Software Productivity Consortium, phone: +1 (703)
742-8877.

Advantages of Inlining S-Functions

Motivations

The goals of generated code usually include compactness and speed. On the other hand, S-functions are runtime-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 runtime 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. TLC 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 datatype
- 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 Chapter 7, “Inlining S-Functions,” 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 . /t1 c_c (for C language targets)
- 4 S-function directory's subdirectory . /t1 c_ada (for Ada 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, Ada, 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
Ada	No	Yes
C	Yes	Yes

Target Language Compiler 4.0 New Features

The following features have been added to the Target Language Compiler for Version 4.0:

- 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. See Appendix A, “model.rtw,” for details. The size of the *model.rtw* file has been reduced.
- Aliases added for block parameters in the *model.rtw* file, see *ParamName0* on page A-53 of the “model.rtw” appendix.
- New flexible methods for text expansion from within strings.
- Column-major ordering of 2-dimensional signal and parameter data.
- New record data handling.
- New TLC language semantics.
- Additional built-in functions: `FIELDNAMES`, `GENERATE_FORMATTED_VALUE`, `GETFIELD`, `ISALIAS`, `ISEMPTY`, `ISEQUAL`, `ISFIELD`, `REMOVEFIELD`, `SETFIELD`
- Support for 2-dimensional signals in inlined code.
- Additional built-in variables: `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 5-37).
- Can expand records with %<>

- Short circuiting of conditionals (`||`, `&&`, `?:`, `%i f-%el sei f-%el se-%endi f`)
- Relational operators can be used with non-finite values.
- Enhanced conversion rules for FEVAL. You can now pass records and structs to FEVAL.

Compatibility Issues

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

- Nested evaluations are no longer supported. Expressions such as
`%<Li bBl ockParameterVal ue(%<myVari abl e>, "", "", "")>`
 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.
- `%codebl ock` and `%endcodebl ock` constructs are no longer supported.
- `%defi nes` & 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 `%i ncl ude` files in different contexts (e.g., from generated files inside of `%wi th` blocks, etc.) `%i ncl ude` statements are now treated as if the 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 `%functi on` in one file and `%i ncl ude` a file that had the `%endfuncti on`). 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 code fragments illustrate the differences between declaring a record `recVar` in previous versions of TLC 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 3-9 for further information.

- Semantics of the `EXISTS` function have changed. In the previous release of TLC, `EXISTS(var)` would check if the variable represented by the string value in `var` existed. In the current release of TLC, `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:

- Chapter 2, “Getting Started,” describes the process that the Target Language Compiler uses to generate code, and general inlining S-function concepts.
- Chapter 3, “Code Generation Architecture,” describes the TLC files and the build process. It also provides a tutorial on how to write target language files.
- Chapter 4, “Contents of `model.rtw`,” describes the `model.rtw` file.
- Chapter 5, “Directives and Built-in Functions,” contains the language syntax for the Target Language Compiler.
- Chapter 6, “Debugging TLC,” explains how to use the TLC debugger.
- Chapter 7, “Inlining S-Functions,” describes how to use the Target Language Compiler and how to inline S-functions.
- Chapter 8, “TLC Tutorial,” contains a selection of examples with step-by-step instructions on how to use the TLC.
- Chapter 9, “TLC Function Library Reference,” contains abstracts for the TLC functions.
- Appendix A, “`model.rtw`,” describes the complete contents of the `model.rtw` file.
- Appendix B, “TLC Error Handling,” lists the error messages that the Target Language Compiler can generate, as well as how to best use the errors.
- Appendix C, “Using TLC with Emacs,” 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 User's Guide* describes the use and internal architecture of the Real-Time Workshop. the “Code Generation and the Build Process” chapter presents information on how TLC fits into the overall code generation process.

- The Simulink manual *Writing S-Functions* 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 the “Guidelines for Writing C MEX S-Functions” chapter, which details how to write wrapped and fully inlined S-functions with a special emphasis on the `mdlRTW()` function.

Getting Started

Code Architecture	2-2
Model.rtw and TLC Overview	2-4
The TLC 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

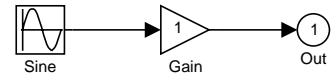
Code Architecture

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

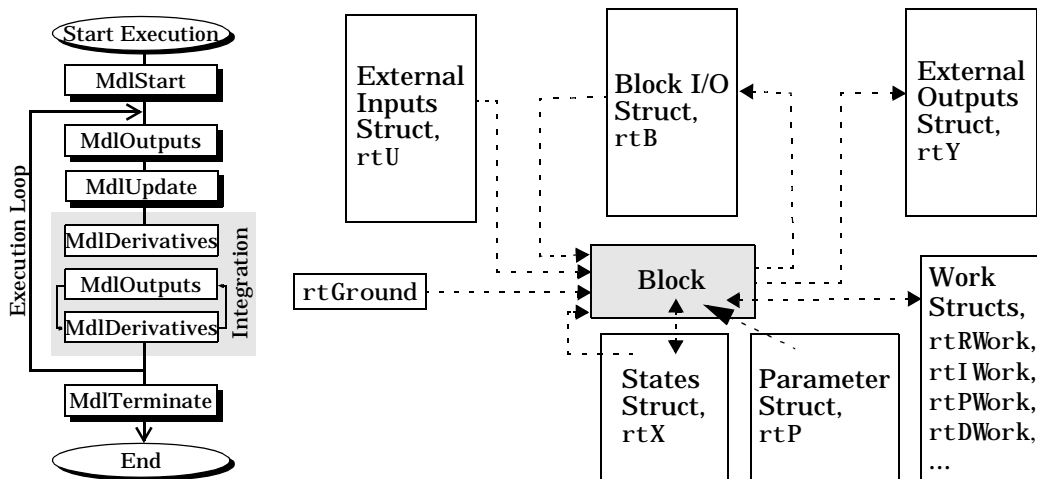
```
void MdlOutputs(int_T tid)
{
    /* Sin Block: <Root>/Sine */
    rtB.Sine = rtP.Sine.Amplitude *
        sin(rtP.Sine.Frequency * ssGetT(rtS) + rtP.Sine.Phase);

    /* Gain Block: <Root>/Gain */
    rtB.Gain = rtB.Sine * rtP.Gain.Gain;

    /* Output Block: <Root>/Out */
    rtY.Out = rtB.Gain;
}
```



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 (rtB). Block inputs can also come from the external input structure (rtU) or the state structure when connected to a state port of an integrator (rtX), or ground (rtGround) if unconnected or grounded. Block outputs can also go to the external output structure (rtY). The following picture 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 TLC-specific pieces of the code generation process.

Model.rtw and TLC Overview

The TLC Process

To write TLC code for your S-function, you need to understand the TLC 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 name/value pairs. The TLC compiler reads the *model.rtw* file and converts it into an internal representation.

Next, the TLC compiler runs (interprets) the TLC files, starting first with the system target file, i.e., *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 TLC 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 name/values 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 apposed 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.

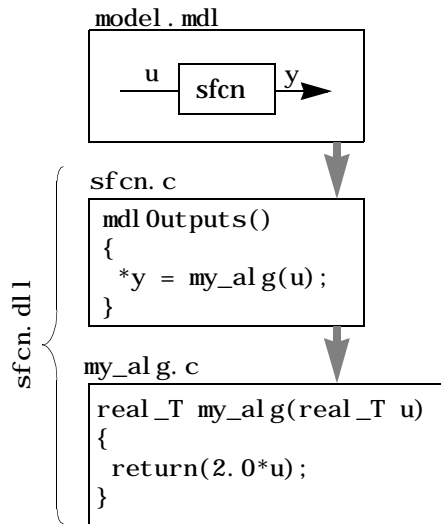
Inlining S-Function Concepts

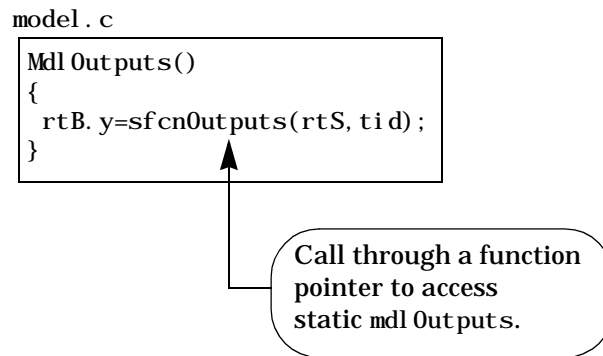
To inline an S-function means to provide a TLC file for an S-function block that will replace the C (or Fortran/M/Ada) code version of the block that was used during simulation.

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

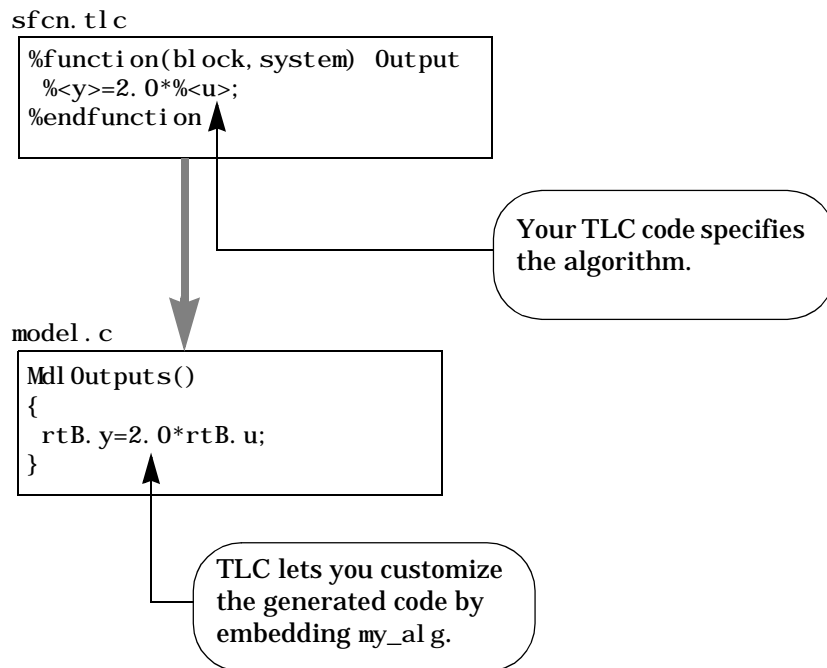
While both effectively inline the S-function and remove the overhead of a noninlined S-function, the two approaches are different. The first example below using `timestwo.tlc` is considered a fully inlined TLC file, where the full implementation of the block is contained in the TLC file for the block.

The second example uses a wrapper TLC file. Instead of generating 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 apposed to each creating identical algorithmic code.
- It provides a way to incorporate legacy C code seamlessly into Real-Time Workshop's 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.

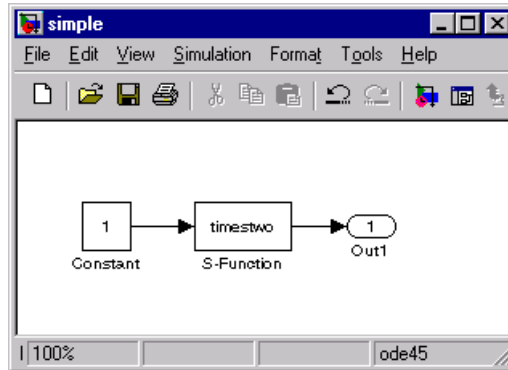
The last piece to the prototype is `Output`. This means that any line that is not a TLC directive is output by the function to the current file that is selected in TLC. So, any nondirective lines in the `Outputs` function become generated code for the block.

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 builtin blocks in *matlabroot/rtw/c/tlc*.

timestwo Model

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



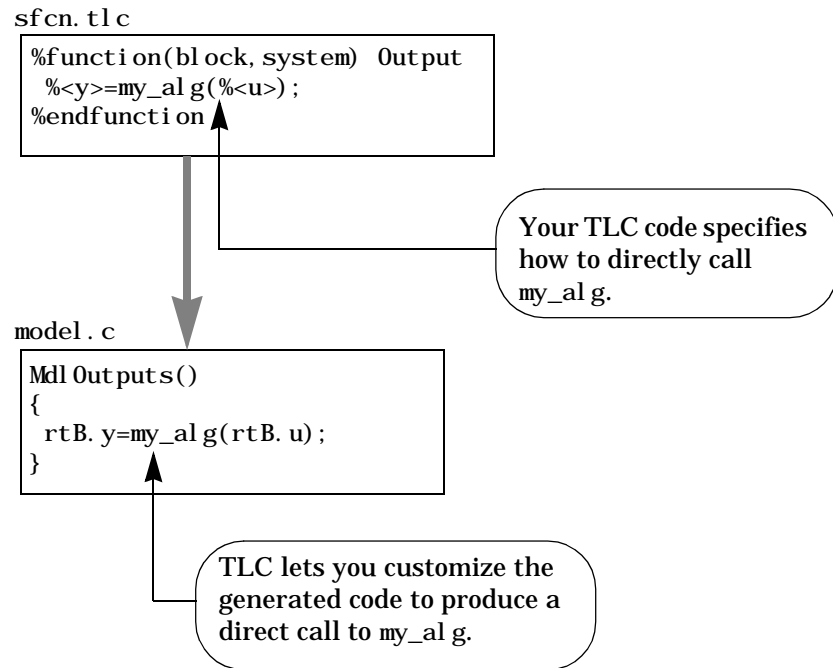
MdlOutputs Code

```
void MdlOutputs(int_T tid)
{
    /* S-Function Block: <Root>/S-Function */
    /* Multiply input by two */
    rtB.S_Funcion = (rtB.Constant_Val ue) * 2.0;

    /* Outport Block: <Root>/Out1 */
    rtY.Out1 = rtB.S_Funcion;
}
```

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 numEl s = LibBlockOutputSignalWidth(0)
/* Multiply input by two */
mytimestwo(%<inPtr>, %<outPtr>, %<numEl s>);

%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, `Mdl Outputs`, then becomes

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

    /* Output Block: <Root>/Out1 */
    rtY.Out1 = rtB.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. Chapter 4, "Contents of `model.rtw`," and Appendix A, "`model.rtw`," contain more details of the `model.rtw` file and its contents. Chapter 7, "Inlining S-Functions," and Chapter 8, "TLC Tutorial," contain details on writing TLC files, including a comprehensive tutorial.

Code Generation Architecture

Build Process	3-2
A Basic Example	3-2
Invoking TLC to Generate Code	3-7
Code Generation Concepts	3-8
Output Streams	3-8
Variable Types	3-9
Records	3-9
Record Aliases	3-11
TLC Files	3-13
Introducing Target Files	3-13
System Target Files	3-15
Block Target Files	3-16
Block Target File Mapping	3-16
Target Files	3-16
Writing Target Language Files: A Tutorial	3-22
Matrix Parameters in Real-Time Workshop	3-22
Configuring TLC	3-25

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.

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

```
%i f 1
    Hello
%el se
    %<Invalid_function_call()>
%endi f
```

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`.

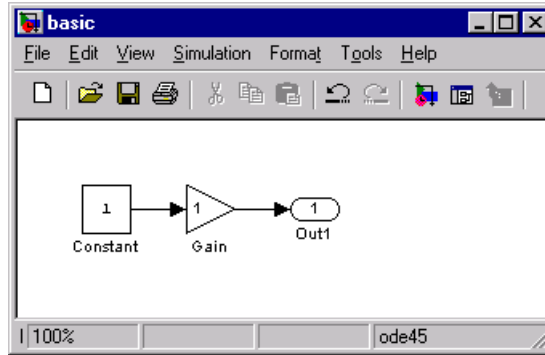


Figure 3-1: Simulink Model

Selecting **Simulation Parameters** from Simulink's **Simulation** menu displays the **Simulation Parameters** dialog box.

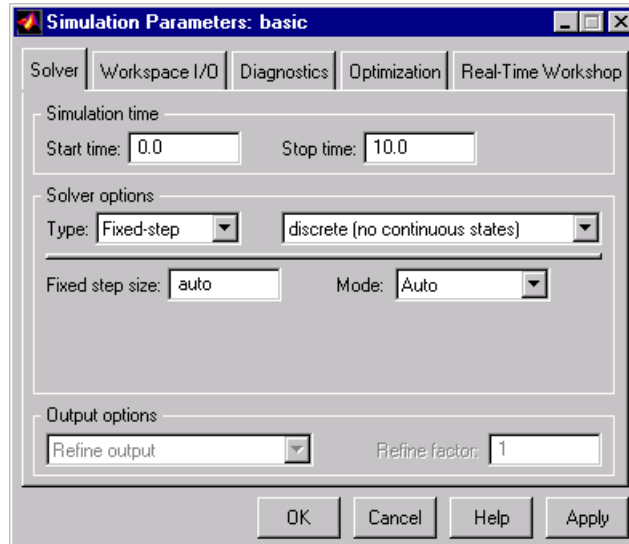


Figure 3-2: Simulation Parameters Dialog Box

Select **Fixed Step Solver** from the **Simulation Parameters** dialog box and then move to the **Real-Time Workshop** tab. Then, click the **Generate Code**

only button. Next, move to the **Category** drop down list and select the **TLC Debugging** option. The tab changes and you should check the **Retain .rtw file** option. Next, click the **Generate Code** button.

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 says:

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

Viewing `basic.rtw`.

Open the file `./basic_grt_rtw/basic.rtw` in a text editor to see what it looks like. The file should look similar to this

```
CompiledModel {
  Name                "basic"
  ...
  <General model information such as>
  Solver              FixedStepDiscrete
  ...
  BlockOutputs {
    -- Information blocks output signals: characteristics and connectivity
    BlockOutputDefaults {
      TestPoint        no
      ...
    }
    ...
    BlockOutput {
      Identifier        Constant
      SigIdx            [0, 1]
      StorageClass      DefinedInTLC
      SigSrc            [0, 0, 0]
    }
  }
  ...
  System {
    <Subsystem description>
    Type                root
    Name                "<Root>"
    ...
    <Blocks in subsystem, listed by block execution order>
    Block {
      Type              Constant
      Name              "<Root>/Constant"
      ...
      Parameters        [1, 1]
      Parameter {
        Name            "Value"
        ...
      }
    }
  }
}
```

```

    }
  }
  ...
}
...
}

```

Creating the Target File. Next, create a `basic.tlc` 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, and `-v` tells TLC to be verbose.

The output of this pair of commands is

```

My model is called basic.
It was generated on Tue May 09 11:41:40 2000.

```

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 TLC to Generate Code

Typically, `rtwgen` and TLC (as seen in the first section) are called directly from the Real-Time Workshop build procedure since the structure and arguments may change from release to release. However, while working with the TLC code, it may be a good idea to call `rtwgen` and TLC directly from the MATLAB prompt.

To generate the `model.rtw` file from the MATLAB prompt, it is typically enough to say

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

However, you may want to add the output option to place the file in the build directory. This generates the `model.rtw` file. However, you may specify other options to `rtwgen` such as whether or not to have case sensitivity for identifiers. For more details, type

```
help rtwgen
```

at the MATLAB prompt.

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
- `-rmodel.rtw`
- `-ai dent=expression`, which assigns a value to the TLC identifier `i dent`. This is discussed in “Configuring TLC” on page 3-26.

For more details, type

```
help tlc
```

at the MATLAB prompt.

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 program 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 program, the `%selectfile` directive tells the Target Language Compiler to send any following text that it 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.

```

%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 Chapter 5, “Directives and Built-in Functions,” for more information on the implicit data types of variables.

Records

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

```

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

```

where `recVar` is the name of the 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 several subrecords. Thus, a simple program that loops through a model and outputs the name of all blocks in the model would look like the following code.

```
%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`.

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

Note Appendix A, “*model.rtw*,” describes in detail the structure of the *model.rtw* file including all the field names and the interpretation of their values.

A record read in 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 `NumNonvrtSubsystems`, `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/*.tlc`. For a complete list of available functions, refer to Chapter 9, “TLC Function Library Reference.”

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 expressions 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.

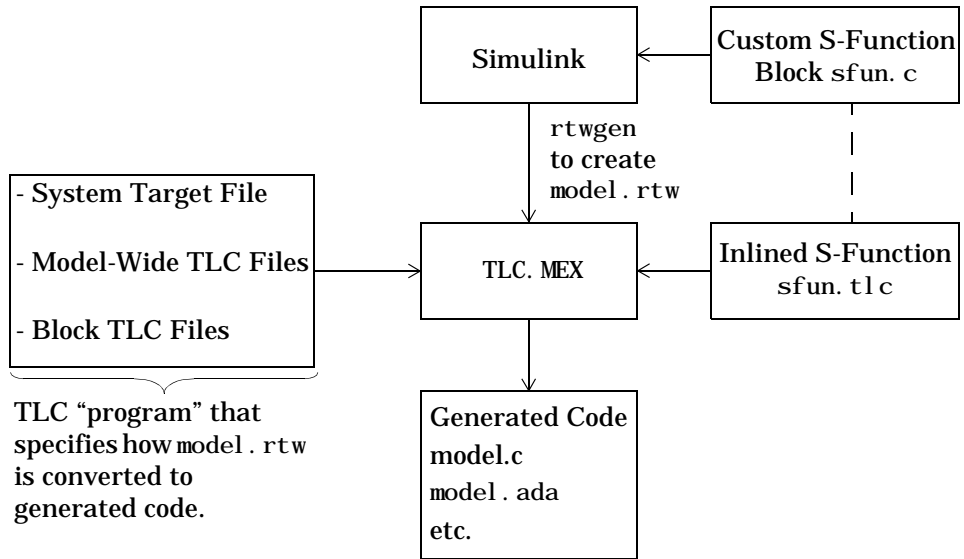
```
%function func(value)
    %createrecord foo { field value }
    %createrecord a { foo foo }
    ISALIAS(a.foo) = %<ISALIAS(a.foo)>
    %return a.foo
%endfunction

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

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. 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*.

Introducing Target Files

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 Chapter 5, “Directives and Built-in Functions,” for a description of the Target Language and Chapter 8, “TLC Tutorial,” for a tutorial on using the Target Language and a description of how 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** page of the **Simulation Parameters** dialog box. A close examination of a system target file reveals how code generation occurs. This is a listing of the non-comment 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 the 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 User's Guide*. 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/genmap.tlc*. All the TLC files listed are located in *matlabroot/rtw/c/tlc* for C and *matlabroot/rtw/ada/tlc* for Ada.

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

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.

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

Appendix A, “model.rtw,” contains a thorough description of the *model.rtw* file, which is useful for creating and/or modifying target files.

Model-Wide Target Files and System Target Files

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

The system target file is the *entry point* for the TLC program, which 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 code generation. *entry.tlc*, which is the entry point into the Real-Time Workshop target files. For a complete list of available system target files for the Real-Time Workshop, see the *Real-Time Workshop User's Guide*.

There are four sets of model-wide target files, one for each of the basic code formats that the Real-Time Workshop supports. These tables list 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 RTW 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 mrlexport.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 BlockOutputs, Parameter, 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 sfclib.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)

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

Model-Wide Target File	Code Format	Purpose
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
sfcnmi d.tlc	RTW S-function	Creates <i>model_sf_mi d.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 mrreg.tlc ertreg.tlc sfcnreg.tlc	Static real-time Malloc real-time Embedded-C RTW S-function	Creates the source file <i>model_reg.h</i> that is included by the <i>model.c</i> file to satisfy the API (all formats)
sfcnsi d.tlc	RTW S-function	Creates <i>model_sf_si d.c</i> , which contains data for an RTW S-function.
srtwi de.tlc mrzwi de.tlc ertwi de.tlc sfcnwi de.tlc	Static real-time Malloc real-time Embedded-C RTW S-function	The entry point for code format. This file produces <i>model.h</i> , <i>model.c</i> , <i>model_reg.h</i> , <i>model_prm.h</i> , <i>model_export.h</i> , and, optionally, <i>model.dt</i> .

Table 3-2: Model-Wide Target Files for Ada

Model-Wide Target File	Purpose
adabody.tlc	Generates the <i>model.adb</i> file, which contains the package body for the model.
adalib.tlc	Contains utility functions used by the other model-wide target files
adaspec.tlc	Generates the <i>model.ads</i> file, which contains the package specification for the model, and the package specifications required for auto-building simulation and real-time targets: <ul style="list-style-type: none"> • <i>register.ads</i> • <i>register2.ads</i> • <i>rt_engine-rto_data.ads</i>
adatypes.tlc	Generates the <i>model_types.ads</i> file, which contains the data structures required by the model. Note that all data declaration instances are declared in the <i>model.adb</i> file.
adawide.tlc	The entry point for Ada code format. It produces these files: <ul style="list-style-type: none"> • <i>model.ads</i> • <i>model.adb</i> • <i>model_types</i> • <i>register.ads</i> • <i>register2.ads</i> • <i>rtengine-rto_data.ads</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.

Writing Target Language Files: A Tutorial

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’s 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 is convenient for allowing 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           "OutputVal ues"
    Val ue         Matrix(3, 3)
    [[1. 0,  2. 0,  3. 0]; [4. 0,  5. 0,  6. 0]; [7. 0,  8. 0,  9. 0];]
    String         "t"
    StringType     "Vari abl e"
    ASTNode {
        IsNonTermi nal      0
        Op                  SL_NOT_I NLI NED
        Model ParameterIdx  3
    }
}

```

and results in this definition in *model.h*

```

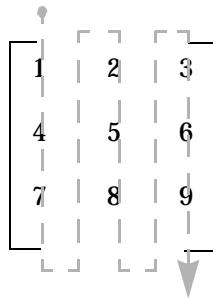
typedef struct Parameters_tag {
    real_T s1_Look_Up_Table_2_D_Table[9];
    /* Variabl e: s1_Look_Up_Table_2_D_Table
     * External Mode Tunabl e: yes
     * Referenced by block:
     * <S1>/Look-Up Table (2-D)
     */

    [ ... other parameter definitions ... ]

} Parameters;

```

The *model_prm.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 rtP = {
  /* 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 matrix parameter access routines,
`Li bBlockMatrixParameter` and `Li bBlockMatrixParameterAddr`, where:

`Li bBlockMatrixParameter(OutputValues, "", "", 0, "", "", 1)` returns
 for C

"rtP. s1_Look_Up_Table_2_D_Table[nRows]" (automatically optimized from
 "[0+nRows*1]")

and

`Li bBlockMatrixParameterAddr(OutputValues, "", "", 0, "", "", 1)`
 returns for C

"&rtP. 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
`Li bBlockParameter` or `Li bBlockParameterAddr`.

Configuring TLC

You can configure TLC from the **RTW Options** dialog box or from the TLC command line, which is also accessible from the **RTW Options** dialog box. To use the **RTW Options** dialog box, select **Tools -> Real-Time Workshop -> Options** from the Simulink menu. Alternatively, you can select **Simulation -> Simulation Parameters** and then select the **Real-Time Workshop** tab from the resulting dialog box.

From the **Category** drop down list, select **TLC Debugging**. This provides options for configuring the build process, including activating the TLC debugger and an option to retain the RTW file. This is covered in more detail in Chapter 6, “Debugging TLC.”

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

```
%assign myVar = 1
```

in your target file. You can repeat the `-a` parameter, and it can be specified in the **System Target File** field in the **Real-Time Workshop** (Category: Target Configuration) 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.

Contents of model.rtw

Overview of model.rtw File	4-2
Using Scopes in the model.rtw File	4-3
Using Library Functions to Access model.rtw Contents	4-6
Caution Against Directly Accessing Record Fields	4-6
Exception to Using the Library Functions	4-7

Overview of model.rtw File

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** page 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-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-value pair

```
      .  
      .  
NumDataOutputPorts 1  
      .  
      .
```

NumDataOutputPorts is the identifier and 1 is its value.

A *record* is specified as

```
RecordName {  
      .  
      .  
}
```

A record contains parameter-value pairs and/or subrecords. For example, this record contains one 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    "model name"
    ...
    System {
        Block {
            Type    "S-Function"
            Name    "<S3>/S-Function"
            ...
            Parameter {
                Name "P1"
                Value Matrix(1,2) [[1, 2];]
            }
        }
        ...
        Block {
        }
    }
    ...
    System {
    }
}
```

- Example of a parameter-value pair (record field).
- There is one system for each nonvirtual subsystem.
- Block records for each nonvirtual block in the 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,

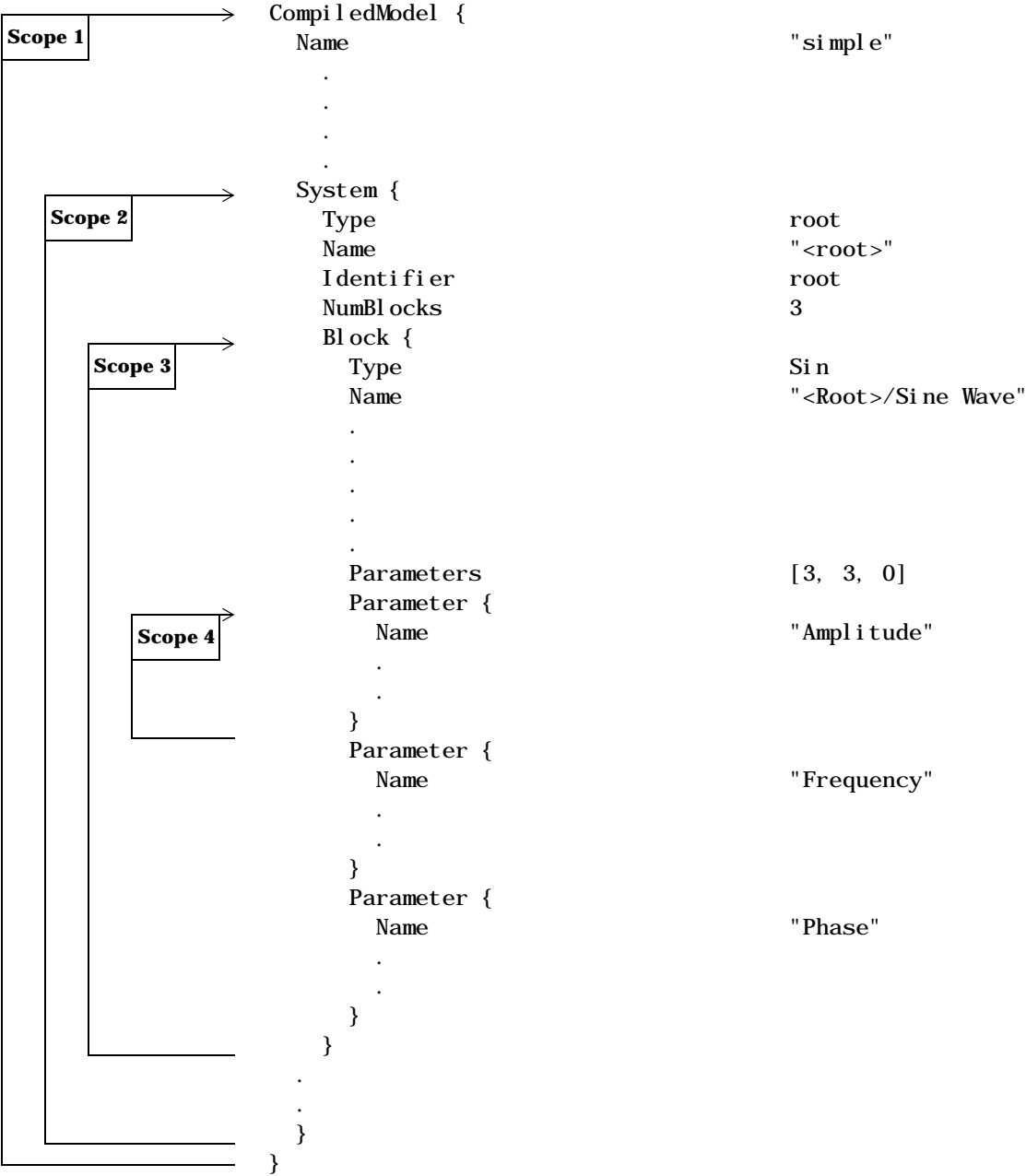
```
%with CompiledModel.System[0].Block[0]
%assign blockName = Name
%endwith
```

`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`.

For a full description of the `model.rtw` file, see Appendix A. Note that the parameter names in this file changes from release to release.



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/values pairs in the block record as apposed to accessing the parameter/values directly from your block TLC code.

See Chapter 9, “TLC Function Library Reference,” 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 ensures it is written to be flexible enough to generate code for any instance/configuration of the block and across releases. An exception is when it is necessary to access directly a field in the Block's record. This is for Parameter Settings, which is discussed below.

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 Chapter 7, “Inlining S-Functions.”

Directives and Built-in Functions

Compiler Directives	5-2
Syntax	5-2
Comments	5-15
Line Continuation	5-16
Target Language Values	5-17
Target Language Expressions	5-19
Formatting	5-25
Conditional Inclusion	5-25
Multiple Inclusion	5-27
Object-Oriented Facility for Generating Target Code	5-32
Output File Control	5-34
Input File Control	5-35
Asserts, Errors, Warnings, and Debug Messages	5-36
Built-In Functions and Values	5-37
TLC Reserved Constants	5-50
Identifier Definition	5-51
Scoping	5-55
Target Language Functions	5-59
 Command Line Arguments	 5-65
Filenames and Search Paths	5-66

Compiler Directives

Syntax

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

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

where *keyword* represents one of the Target Language Compiler's directives, and [*argument1*, *argument2*, ...] represents expressions that define any required parameters. For example,

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

uses the `%assign` directive to 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. Beginning a line with `%%` lets you include a comment on a line.

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

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

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>%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>%assign x = "%<expr>"</pre> <p>Evals outside of strings, such as the following example, should be avoided whenever possible.</p> <pre>%assign x = %<expr></pre>

Directive	Description
<pre>%i f expr %el sei f expr %el se %endi f</pre>	<p>Conditional inclusion, where the constant-expression <i>expr</i> must evaluate to an integer. It is not necessary to expand variables or expressions using the %<<i>expr</i>> notation. Expanding the expression may be necessary when the expression needs to be re-evaluated. For example, assume <i>k</i> represents the value of a the gain block parameter which may be a number or a string variable. The following code will check if <i>k</i> is the numeric value 0.0 by executing a TLC library function to check for equality. Notice that the %<<i>expr</i>> syntax isn't used since we are operating on a directive line. Like other languages, expression evaluation do short circuit.</p> <pre>%i f ISEQUAL(k, 0.0) <text and directives to be processed if, k is 0.0> %endi f</pre>
<pre>%swi t ch expr %case expr %break %default %break %endswi t ch</pre>	<p>The <i>swi t ch</i> directive is very similar to the C language <i>swi t ch</i> statement. The expression, <i>expr</i>, can be of any type that can be compared for equality using the == operator. If the %break is not included after a %case statement, then it will fall through to the next statement.</p>
<pre>%wi th %endwi th</pre>	<p><i>model</i>.rtw is organized as a hierarchy of records. The</p> <pre>%wi th recordName %endwi th</pre> <p>directives let you make a given record the current scope.</p>
<pre>%setcommandswi t ch</pre>	<p>Changes the value of a command-line switch as in</p> <pre>%setcommandswi t ch "-v1"</pre>
<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 the command line option "-da" has to be added.</p>

Directive	Description
<code>%error</code> <code>%warning</code> <code>%trace</code> <code>%exit</code>	<p>Flow control directives:</p> <p><code>%error tokens</code> — The <i>tokens</i> are expanded and displayed.</p> <p><code>%warning tokens</code> — The <i>tokens</i> are expanded and displayed.</p> <p><code>%trace tokens</code> — The <i>tokens</i> are expanded and displayed only when the “verbose output” command line option <code>-v</code> or <code>-v1</code> is specified.</p> <p><code>%exit tokens</code> — The <i>tokens</i> are expanded, displayed, and TLC exits.</p> <p>Note, when reporting errors, you should use</p> <p><code>%exit Error Message</code></p> <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> <p><code>%setcommandswitch "-v1" %% force TLC stack trace</code></p> <p><code>%exit Assert message</code></p>
<code>%assign</code>	<p>Creates identifiers (variables). The general form is</p> <p><code>%assign [::] variable = expression</code></p> <p>The <code>::</code> 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> <p><code>%assign nameInfo = "The name of this is %<Name>"</code></p> <p>or alternately</p> <p><code>%assign nameInfo = "The name of this is " + Name</code></p> <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 3-11.</p>

Directive	Description
<code>%createrecord</code>	<p>Creates records. This command accepts a list of one or more record specifications (e.g., <code>{ foo 27 }</code>). Each record specification contains a list of zero or more name-value pairs (e.g, <code>foo 27</code>) that become the members of the record being created, where the values it selves can be record specifications.</p> <pre>%createrecord NEW_RECORD { foo 1 ; SUB_RECORD {foo 2} } %assign x = NEW_RECORD. SUB_RECORD. foo</pre> <p>If more than one record specification appears, these additional record specifications construct an array of records.</p> <pre>%createrecord RECORD_ARRAY { NEW_RECORD { foo 1 } } ... { NEW_RECORD { foo 2 } } ... { NEW_RECORD { foo 3 } } %assign x = RECORD_ARRAY[1]. NEW_RECORD. foo</pre> <p>If <code>::</code> is the first token after he <code>%createrecord</code> token, the record is created in the global scope.</p>
<code>%addtorecord</code>	<p>Adds fields to an existing record. The new fields may be name-value pairs or aliases to already existing records.</p> <pre>%addtorecord OLD_RECORD foo 1</pre> <p>If the new field being added is a record, then <code>%addtorecord</code> will make an alias to that record instead of a deep copy. To make a deep copy, use <code>%copyrecord</code>.</p> <pre>%createrecord NEW_RECORD { foo 1 } %addtorecord OLD_RECORD NEW_RECORD_ALIAS NEW_RECORD</pre>

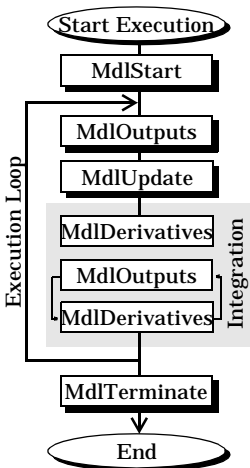
Directive	Description
<code>%mergerecord</code>	<p>Adds (or merges) one or more records into another. The first record will contain the results of the merge of the first record plus the contents of all the other records specified by the command. The contents of the second (and subsequent) records are deep copied into the first (i.e., they are not references).</p> <pre>%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>
<code>%copyrecord</code>	<p>Makes a deep copy of an existing record. It creates a new record in a similar fashion to <code>%createrecord</code> except the components of the record are deep copied from the existing record. Aliases are replaced by copies.</p> <pre>%copyrecord NEW_RECORD OLD_RECORD</pre>
<code>%real format</code>	<p>Specifies how to format real variables. To format in exponential notation with 16 digits of precision, use</p> <pre>%real format "EXPONENTIAL"</pre> <p>To format without loss of precision and minimal number of characters, use</p> <pre>%real format "CONCISE"</pre> <p>When inlining S-functions, the format is set to <code>conci se</code>. You can switch to <code>exponenti al</code>, but should switch it back to <code>conci se</code> when done.</p>
<code>%l anguage</code>	<p>This must appear before the first <code>GENERATE</code> or <code>GENERATE_TYPE</code> function call. This specifies the name of the language as a string, which is being generated as in <code>%l anguage "C"</code>. Generally, this is added to your system target file.</p>

Directive	Description
<code>%implements</code>	<p>Placed within the <code>.tlc</code> file for a specific record type, when mapped via <code>%generatefile</code>. The syntax is: <code>%implements "Type" "Language"</code>. When inlining an S-function in C, this should be the first non-comment line in the file as in</p> <pre>%implements "s_function_name" "C"</pre> <p>The next noncomment lines will be <code>%function</code> directives specifying the functionality of the S-function.</p> <p>See the <code>%language</code> and <code>GENERATE</code> function descriptions for further information.</p>
<code>%generatefile</code>	<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 <code>.tlc</code> file that implements the functionality of the block as in</p> <pre>%generatefile "Sin" "sin_wave.tlc"</pre>
<code>%filescope</code>	<p>Limits the scope of variables to the file in which they are defined. All variables defined after the appearance of <code>%filescope</code> in a file have this property, otherwise they default to global variables.</p> <p><code>%filescope</code> is useful in conserving memory. Variables whose scope is limited by <code>%filescope</code> 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>
<code>%include</code> <code>%addincludepath</code>	<p><code>%include "file.tlc"</code> — insert specified target file at the current point. Use <code>%addincludepath "directory"</code> to add additional paths to be searched. Be sure to escape backslashes in PC directory names as in <code>"C:\\mytlc"</code>. Note that <code>%include</code> directives behave as if they were in a global context.</p>

Directive	Description
<code>%roll</code> <code>%endroll</code>	<p>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 <code>%roll</code> directive is for a gain operation, $y=u*k$</p> <pre> 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>The <code>%roll</code> directive is similar to <code>%foreach</code>, except it iterates the identifier (<code>sigIdx</code> 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 <code>RollRegions</code> is automatically computed and placed in the <code>Block</code> record. An example of a roll regions vector is <code>[0: 19, 20: 39]</code>, where we have two contiguous ranges of signals passing through the block. The first is <code>0: 19</code> and the second is <code>20: 39</code>. 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>Each time through the <code>%roll</code> loop, <code>sigIdx</code> 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 <code>RollThreshold</code> 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., <code>"i"</code>), otherwise it will be <code>""</code>.</p>

Directive	Description
<code>%roll</code> (continued)	<p>The <code>block</code> parameter is the current block that is being rolled. The "Roller" parameter specifies the name for an internal <code>GENERATE_TYPE</code> 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>. "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>. "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 <i>name</i>. 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 <i>name</i>. 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 <i>name</i>. 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 <i>name</i>. 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 <i>name</i>. "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.

Directive	Description
<code>%roll</code> (continued)	<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 <i>name</i>,</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, <i>sigIdx</i> and <i>lcv</i> are local to the body of the loop.</p>

Directive	Description
<code>%function</code> <code>%return</code> <code>%endfunction</code>	<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>
	<p>For block target files, you can add to your inlined .tlc 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 BlockInstanceData(block, system) Output Called once during model initialization to place code in the registration function (<i>model_reg.h</i>) to allocate persistent data. %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 Mdl Start.</pre>

Directive	Description
%function %return %endfunction <i>(continued)</i>	<p>%function InitializeConditions(block, system) Output Use this function if your block has state that needs to be initialized at the start of execution and when an enabled subsystem resets states. Place in Mdl Start and/or subsystem initialization routines.</p> <p>%function Outputs(block, system) Output The primary function of your block. Place in Mdl Outputs.</p> <p>%function Update(block, system) Output Use this function if your block has actions to be performed once per simulation loop, such as updating discrete states. Place in Mdl Update</p> <p>%function Derivatives(block, system) Output Used this function if your block has derivatives for Mdl Derivatives.</p> <p>%function Terminate(block, system) Output Use this function if your block has actions that need to be in Mdl Terminate.</p>
%foreach %endforeach	<p>Multiple inclusion that iterates from 0 to the upperLimit - 1 constant integer expression. Each time through the loop, the loopIdentifier, (e.g., x) is assigned the current iteration value.</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 upperLimit expression is cast to a TLC integer value. The loopIdentifier is local to the loop body.</p>

Directive	Description
<code>%for</code>	<p>Multiple inclusion directive with syntax</p> <pre>%for i dent 1 = const-exp1, const-exp2, i dent2 = const-exp3 %body %break %cont i nue %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>%cont i nue</code> directives act the same as they do in the <code>%foreach</code> directive. <i>const-exp2</i> is a Boolean expression that indicates whether the loop should be rolled (see <code>%rol l</code> above).</p> <p>If <i>const-exp2</i> evaluates to TRUE, <i>i dent2</i> is assigned the value of <i>const-exp3</i>. Otherwise, <i>i dent2</i> is assigned an empty string.</p> <p>Note: <i>i dent1</i> and <i>i dent2</i> above are local to the loop body.</p>

Directive	Description
<code>%openfile</code> <code>%selectfile</code> <code>%closefile</code>	<p>These are used to manage the files that are created. The syntax is</p> <pre> %openfile <i>streamId</i>="filename.ext" mode {open for writing} %selectfile <i>streamId</i> {select an open file} %closefile <i>streamId</i> {close an open file} </pre> <p>Note that the “<i>filename.ext</i>” is optional. If not specified, <i>streamId</i> and <i>filename</i> will be the same and a variable (string buffer) is created containing the output. The mode argument is optional. If specified, it can be “a” for appending, “r” for reading, or “w” for writing.</p> <p>Note that the special <i>streamId</i> <code>NULL_FILE</code> specifies that no output occur. The special <i>streamId</i> <code>STDOUT</code> specifies output to the terminal.</p> <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 <code>buffer</code> contains the expanded text specified between the <code>%openfile</code> and <code>%closefile</code> directives.</p>
<code>%generate</code>	<p><code>%generate <i>blk fn</i></code> is equivalent to <code>GENERATE(<i>blk</i>, <i>fn</i>)</code>.</p> <p><code>%generate <i>blk fn type</i></code> is equivalent to <code>GENERATE(<i>blk</i>, <i>fn</i>, <i>type</i>)</code>.</p> <p>See “<code>GENERATE</code> and <code>GENERATE_TYPE</code> Functions” on page 5-33.</p>
<code>%undef</code>	<p><code>%undef <i>var</i></code> removes the variable <i>var</i> from scope. If <i>var</i> is a field in a record, <code>%undef</code> removes that field from the record. If <i>var</i> is a record array, <code>%undef</code> 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 5-59.

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, \
    "Roller", rollVars
```

or

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

Note Use \ to suppress line feeds to the output and the ellipsis (...) to indicate line continuation. Note that \ and the ellipsis (...) cannot be used inside strings.

Target Language 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.

Table 5-1: 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 <i>model.rtw</i> 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.

Table 5-1: Target Language Values (Continued)

Value Type String	Example	Description
"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 (i nt on the target machine)
"Range"	[1: 5]	A range of integers between 1 and 5, inclusive
"Real "	3. 14159	A floating-point number (doubl e on the target machine), including exponential notation
"Real 32"	3. 14159F	A 32-bit single-precision floating-point number (fl oat on the target machine)
"Scope"	Block { ... }	A block record
"Speci al "	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\>>.
"Unsi gned"	15U	A 32-bit unsigned integer (unsi gned i nt on the target machine)

Table 5-1: Target Language Values (Continued)

Value Type String	Example	Description
"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">
```

Note It is not necessary to place expressions in the `%< >` format when they appear on directive lines. Doing so causes a double evaluation.

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,

```
%i f 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 5-24 for information on the promotions that result when the Target Language Compiler operates on mixed types of expressions.

Table 5-2: 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

Table 5-2: Target Language Expressions (Continued)

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

Table 5-2: Target Language Expressions (Continued)

Expression	Definition
<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.
<code>expr + expr</code>	<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>
<code>expr - expr</code>	Subtracts the two expressions; the operands must be numeric.
<code>expr << expr</code>	Left shifts the left operand by an amount equal to the right operand; the arguments must be integral.
<code>expr >> expr</code>	Right shifts the left operand by an amount equal to the right operand; the arguments must be integral.

Table 5-2: Target Language Expressions (Continued)

Expression	Definition
<code>expr > expr</code>	Tests if the first expression is greater than the second expression; the arguments must be numeric.
<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.

Table 5-2: Target Language Expressions (Continued)

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

Note Relational operators (<, <=, >, >=, !=, ==) can be used with non-finite values.

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).

	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:

`%real format 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 `%real format` directive sets the default format for Real number output to the selected style for the remainder of processing or until it encounters another `%real format` directive.

Conditional Inclusion

The conditional inclusion directives are

`%if constant-expression`

```
%el se  
%el sei f constant-expression  
%endi f
```

and

```
%swi tch constant-expression  
%case constant-expression  
%break  
%default  
%endswi tch
```

%if

The *constant-expression* must evaluate to an integral expression. It controls the inclusion of all the following lines until it encounters a `%el se`, `%el sei f`, or `%endi f` 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 `%i f` directive are included up until the `%endi f`, `%el sei f`, or `%el se` directives.

When the Compiler encounters an `%el sei f` directive, and no prior `%i f` or `%el sei f` directive has evaluated to nonzero, the Compiler evaluates the expression. If the value is 0, the lines following the `%el sei f` directive are not included. If the value is nonzero, the lines following the `%el sei f` directive are included up until the subsequent `%el se`, `%el sei f`, or `%endi f` directive.

The `%el se` directive begins the inclusion of source text if all of the previous `%el sei f` statements or the original `%i f` statement evaluates to 0; otherwise, it prevents the inclusion of subsequent lines up to and including the following `%endi f`.

The *constant-expression* can contain any expression specified in “Target Language Expressions” on page 5-19.

%switch

The `%swi tch` 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 `%swi tch`, `%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 "Si n"
    /* Matches Si n or falls through from case x. */
    %break
    /* Exits the switch. */
%case "gai n"
    /* Matches gai n. */
    %break
%default
    /* Does not match x, "Si n," or "gai n." */
%endswi tch

```

In general, this is a more readable form for the `%i f/%el sei f/%el se` construction.

Multiple Inclusion

`%foreach`

The syntax of the `%foreach` multiple inclusion directive is

```

%foreach i denti fi er = constant-expression
    %break
    %cont i nue
%endforeach

```

The `constant-expression` must evaluate to an integral expression, which then determines the number of times to execute the `foreach` loop. The `i denti fi er` 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 `%cont i nue` are optional directives that you can include in the `%foreach` directive:

- `%break` can be used to exit the nearest enclosing `%for`, `%foreach`, or `%swi tch` statement.
- `%cont i nue` 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-specific vectors that vastly simplify the body of the `%roll` statement. These functions include `Li bBlockInputSignal`, `Li bBlockOutputSignal`, `Li bBlockParameter`, `Li bBlockRWork`, `Li bBlockIWork`, `Li bBlockPWork`, and `Li bDeclareRollVars`, `Li bBlockMatrixParameter`, `Li bBlockParameterAddr`, `Li bBlockContinuousState`, and `Li bBlockDiscreteState` function reference pages in Chapter 9, “TLC Function Library Reference.” 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 the 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., "Si n" or "Gai n". 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 "Si n" the Compiler would search for "Si n.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 "Si n" "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 "Si n" [ "Ada", "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", "Ada"]
```

Finally, you can implement several types using the wildcard (*) for the type field

```
%implements * ["C", "Ada"]
```

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 `%with` in “Scoping”. 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

```
%addi ncl udepath "C: \di rectory1\di rectory2"      (PC)
%addi ncl udepath "/di rectory1/di rectory2"          (UNIX)
```

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

```
%addi ncl udepath ". \di rectory2"                  (PC)
%addi ncl udepath ". /di rectory2"                   (UNIX)
```

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 %addi ncl udepath 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 %addi ncl udepath directives
- 3 Any include paths specified at the command line via -I

Typically, %addi ncl udepath directives should be specified in your system target file. Multiple %addi ncl udepath 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
%warni ng tokens
%trace tokens
%exi t 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 5-65 for additional information about switches.

The `%assert` directive will evaluate the expression (if you specify the switch `-da` to the Target Language Compiler) and will produce a stack trace if the expression evaluates to a boolean false.

The `%exit` directive reports an error and stops further compilation.

Built-In Functions and Values

Table 5-3, TLC Built-in Functions and Values, 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.

Table 5-3: TLC Built-in Functions and Values

Built-In Function Name	Expansion
CAST(<i>expr</i> , <i>expr</i>)	<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> <p>CAST("Real ", <i>variable-name</i>)</p> <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 %real format).</p>
EXISTS(<i>var</i>)	<p>If the <i>var</i> identifier is not currently in scope, the result is TLC_FALSE. If the identifier is in scope, the result is TLC_TRUE. <i>var</i> can be a single identifier or an expression involving the . and [] operators.</p> <p>Note: prior to TLC release 4, the semantics of EXISTS differed from the above. See "Compatibility Issues" on page 1-18.</p>

Table 5-3: TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
FEVAL(matlab-command, TLC-expressions)	<p>Performs an evaluation in MATLAB. For example</p> <pre>%assign result = FEVAL("sin", 3.14159)</pre> <p>The %matlab 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>
FILE_EXISTS(expr)	<p>expr must be a string. If a file by the name expr does not exist on the path, the result is TLC_FALSE. If a file by that name exists on the path, the result is TLC_TRUE.</p>
FORMAT(real value, format)	<p>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.</p>
FIELDNAMES(record)	<p>Returns a array of strings containing the record field names associated with the record.</p>

Table 5-3: TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
GETFIELD(<i>record</i> , <i>fieldname</i>)	Returns the contents of the specified field name, if the field name is associated with the record.
GENERATE(<i>record</i> , <i>function-name</i> , ...)	This is used to execute function calls mapped to a specific record type (i.e., Block record functions). For example, used 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(<i>type</i>)	For the specified record type, does a .t1c file exist? Use this to see if the GENERATE_TYPE call will succeed.
GENERATE_FORMATTED_VALUE (<i>expr</i> , <i>string</i>)	Returns a potentially multiline string that can be used to declare the value(s) of <i>expr</i> in the current target language (C or Ada). 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.
GENERATE_FUNCTION_EXISTS (<i>record</i> , <i>function-name</i>)	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.

Table 5-3: TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
GENERATE_TYPE (<i>record</i> , <i>function-name</i> , <i>type</i> , ...)	Similar to GENERATE, except <i>type</i> 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 (<i>record</i> , <i>function-name</i> , <i>type</i>)	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, r, I, a
IDNUM(expr)	expr must be a string. The result is a vector where the first element is a leading string (if any) and the second element is a number appearing at the end of the input string. For example, IDNUM("ABC123") yields ["ABC", 123]
IMAG(expr)	Returns the imaginary part of a complex number.
INT8MAX	127
INT8MIN	-128
INT16MAX	32767
INT16MIN	-32768
INT32MAX	2147483647
INT32MIN	-2147483648

Table 5-3: TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
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(<i>expr1</i> , <i>expr2</i>)	Returns TLC_TRUE if the first expression has the same data type and contains the same value than the second expression, and TLC_FALSE otherwise.
ISEMPTY(<i>expr</i>)	Returns TLC_TRUE if the expression contains (resolves to) 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(<i>expr</i>)	Returns TLC_TRUE if the value of the expression is <code>inf</code> and TLC_FALSE otherwise.
ISNAN(<i>expr</i>)	Returns TLC_TRUE if the value of the expression is <code>NAN</code> , and TLC_FALSE otherwise.
ISFINITE(<i>expr</i>)	Returns TLC_TRUE if the value of the expression is not <code>+/- inf</code> or <code>NAN</code> , and TLC_FALSE otherwise.

Table 5-3: TLC Built-in Functions and Values (Continued)

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

Table 5-3: 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>].
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.

Table 5-3: TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
SYSNAME(expr)	<p>Looks for specially formatted strings of the form <code><x>/y</code> and returns <code>x</code> and <code>y</code> as a 2-element string vector. This is used to resolve subsystem names in Real-Time Workshop. For example,</p> <pre>%<sysname(" <sub>/Gai n") ></pre> <p>returns</p> <pre>[" sub" , "Gai n"]</pre> <p>In Block records, the Name of the block is written similar to <code><sys/blockname></code> where <code>sys</code> is <code>S#</code> or <code>Root</code>. You can obtain the full path name by calling <code>LibGetBlockPath(block)</code>; this will include newlines and other troublesome characters that cause display difficulties. To obtain a full path name suitable for one line comments but not identical to the Simulink path name, use <code>LibGetFormattedBlockPath(block)</code>.</p>
TLCFILES	Returns a vector containing the names of all the target files included thus far in the expansion. 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.

Table 5-3: TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
TLC_VERSION	The version and date of the Target Language Compiler.
TYPE(expr)	Evaluates <i>expr</i> 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 Target Language Values table for possible values.
UI NT8MAX	255U
UI NT16MAX	65535U
UI NT32MAX	4294967295U
UI NTMAX	Maximum unsigned integer value on target machine.
WHI TE_SPACE(expr)	Accepts a string and returns 1 if the string contains only whitespace characters (, \t, \n, \r); returns 0 otherwise.
WI LL_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-hand-side argument is allowed when calling MATLAB.

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

TLC Type	MATLAB Type
"Boolean"	Boolean (scalar or Matrix)
"Number"	Double (scalar or Matrix)
"Real "	Double (scalar or Matrix)
"Real 32"	Double (scalar or Matrix)
"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

TLC Type	MATLAB Type
"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 2 dimensions is not supported,

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)
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)

MATLAB Type	TLC Type
Double precision	"Real" (scalar or Matrix)
Complex double precision	"Complex" (scalar or Matrix)
INT64, UINT64	"Double" (scalar or Matrix)
Sparse matrix	Expanded out to matrix of Reals
Cell array of structures	Record array
Cell array of non-structures	Vector or matrix of types converted from the types of the elements
Cell array of structures and non-structures	Conversion not supported
Structure	"Record"
Object	Conversion not supported

Other value types are not currently supported.

As an example, this statement uses the FEVAL built-in function to call MATLAB to take the sine of the input argument.

```
%assign result = FEVAL( "sin", 3.14159 )
```

Variables (identifiers) can take on the following constant values. Note the suffix on the value one.

Constant Form	TLC Type
1.0	"Real "
1.0[F/f]	"Real 32"
1	"Number"
1[U u]	"Unsigned"
1.0i	"Complex"

Constant Form	TLC Type
1[Ui ui]	"Unsi gnedGaussi an"
1i	"Gaussi an"
1. 0[Fi fi]	"Compl ex32"

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	+i nf
rtMi nusInf	- i nf
rtNaN, NaN, nan	nan
rtInfi, Infi, infi	i nf*i
rtMi nusInfi	- i nf*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, rtMi nusInf, - i nf, rtNaN, nan
and their corresponding version when complex

rtInfi, infi, rtMi nusInfi, - i nfi, rtNaNi

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-hand 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 `::` operator to disambiguate, as in

```
%function ...
%assign foo = 3
%assign ::foo = foo
```

```
...  
%endfunction
```

The `::` 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.

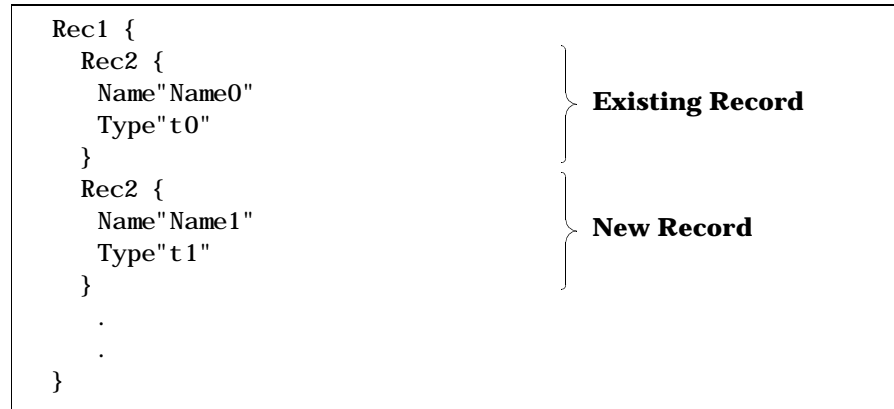


Figure 5-1: Adding a New Record

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

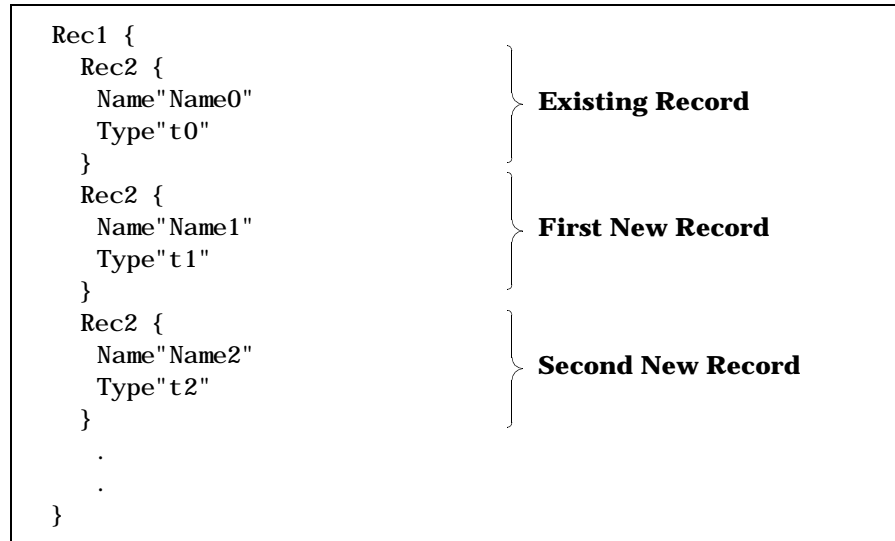


Figure 5-2: Adding Multiple Records

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.

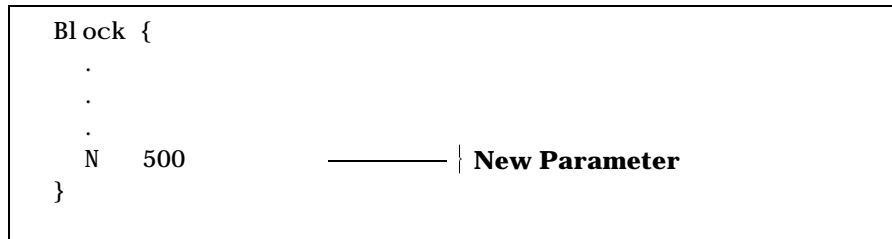


Figure 5-3: Parameter Added to Existing Record

Scoping

The structure of the `%with` directive is

```

%with expression
%endwith
  
```

The `%with` directive adds a new scope to be searched onto the current list of scopes. This directive makes it easier to refer to block-scoped variables.

For example, if you have the following Real-Time Workshop file

```

System {
    Name "foo"
}
  
```

You can access the `Name` parameter without a `%with` statement, by using

```
%<System. Name>
```

or by using `%with`

```

%with System
    %<Name>
%endwith
  
```

Variable Scoping

The Target Language Compiler uses dynamic scoping to resolve references to variables. This section illustrates how the Target Language Compiler determines the values of variables.

In the simplest case, to resolve a variable the Target Language Compiler searches the top-level Real-Time Workshop pool followed by the global pool. This illustration shows the search sequence that the Target Language Compiler uses.

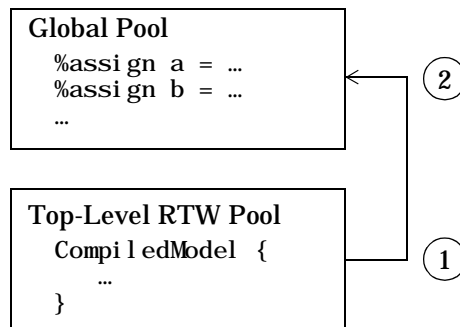


Figure 5-4: Search Sequence

You can modify the search list and search sequence by using the `%with` directive.

Example

When you add the following construct

```

%with CompiledModel.System[sysidx]
...
%endwith
  
```

the `System[sysidx]` scope is added to the search list, and it is searched before anything else.

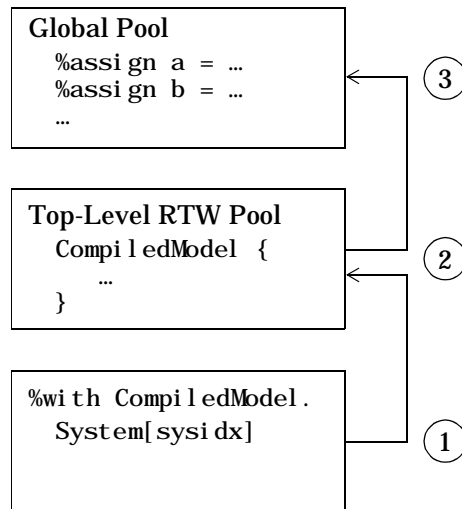


Figure 5-5: Modifying the Search Sequence

Using this technique makes it simpler to access embedded definitions. For example, to refer to the system name without using `%with`, you would have to use

```
CompiledModel . System[sysidx] . Name
```

Using the `%with` construct (as in the previous example), you can refer to the system name simply by

```
Name
```

The rules within functions behave differently. A function has its own scope, and that scope gets added to the previously described list as depicted in this figure.

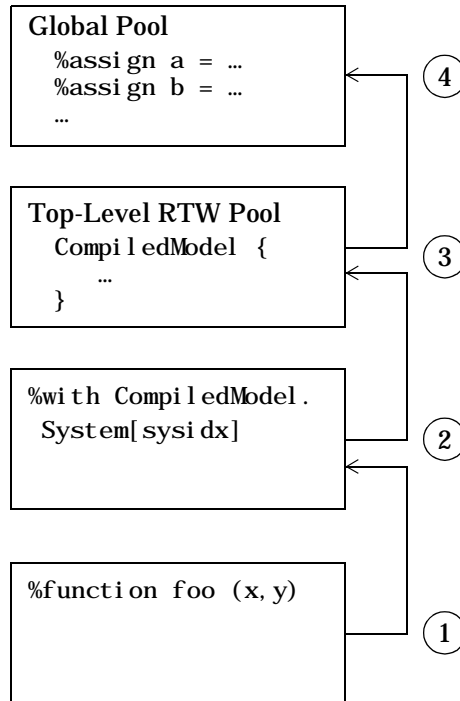


Figure 5-6: Scoping Rules Within Functions

For example, if you have the following code,

```

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

```

.
.
%function foo (a, b)
.
.
.
    assign myvar=Name
.
.
.
%endfunction

```

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

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 Target Language Values table.

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 new entry in the function's block within the scope chain, and does not affect any of the other entries. That is, it is local to the function. For example,

```
%function foo (x, y)
%assign local = 3
%endfunction
```

adds `local = 3` to the `foo()` block in the scope list giving

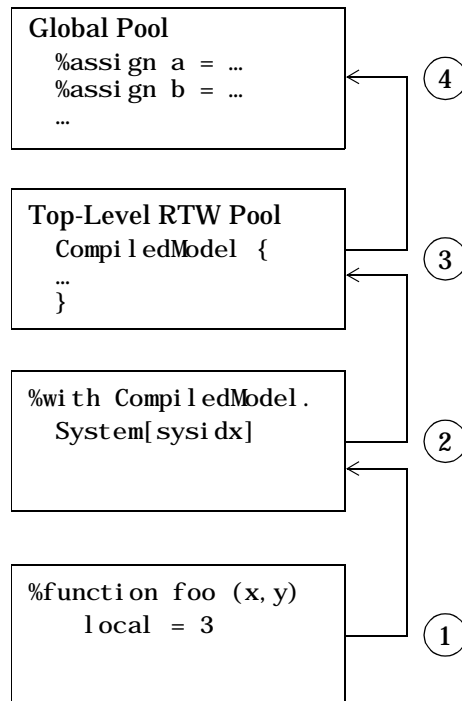


Figure 5-7: Scoping Rules Within Functions Containing Local Variables

You can override this default behavior by using `%assign` with the `::` operator. For example,

```
%assign :: global = 3
```

makes `global` a global variable and initializes it to 3.

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.

For example,

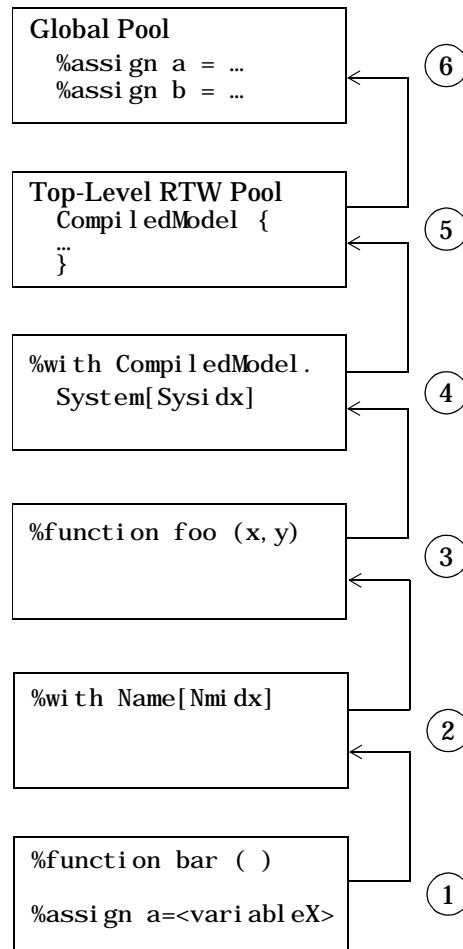


Figure 5-8: Scoping Rules When Using %with Within a Function

%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.

Table 5-4: 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 verbose level to <number>. Omitting this option sets the verbose 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.

Table 5-4: Target Language Compiler Switches (Continued)

Switch	Meaning
-d[a n g o]	Specifies the level and type of debugging. By default, debugging is off (-do). -d defaults to -dn, or normal mode debugging, -dg is generate mode debugging and -da switches on the assertion evaluation.
-a[i dent]=expr	Specifies an initial value, <expr>, for the identifier, <i dent>, for some parameters; equivalent to the %assi gn 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 .tlc extension. By default, block-level files have the same name as the Type of the block in which they appear. You can override the search path for target files with your own local versions. The Target Language Compiler finds all target files along this path. If you specify additional search paths with the -I switch of the tlc command or via the %addi ncl udepath directive, they will be searched after the current working directory, and in the order in which you specify them.

Debugging TLC

About the TLC Debugger	6-2
Tips for Debugging TLC Code	6-2
Using the TLC Debugger	6-3
Tutorial	6-3
TLC Debugger Commands	6-6
Example TLC Debugging Session	6-9
TLC Coverage	6-11
Using the TLC Coverage Option	6-11
TLC Profiler	6-14
Using the Profiler	6-14

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.

```
%set command switch "-v1"
```

- 2 To trace the value of a variable in a function, place the following statement in your TLC file.

```
%warning This is in my function %<variable>
```

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

Using the TLC Debugger

This section provides a tutorial that illustrates the steps for invoking and using the TLC debugger with Simulink and RTW generated code. The files and models for this example are in

`matlabroot/toolbox/rtw/rtwdemos/tlctutorial/tldebug/`

This tutorial uses the `simple_log` model and the `gain.tlc` TLC file located in this directory.

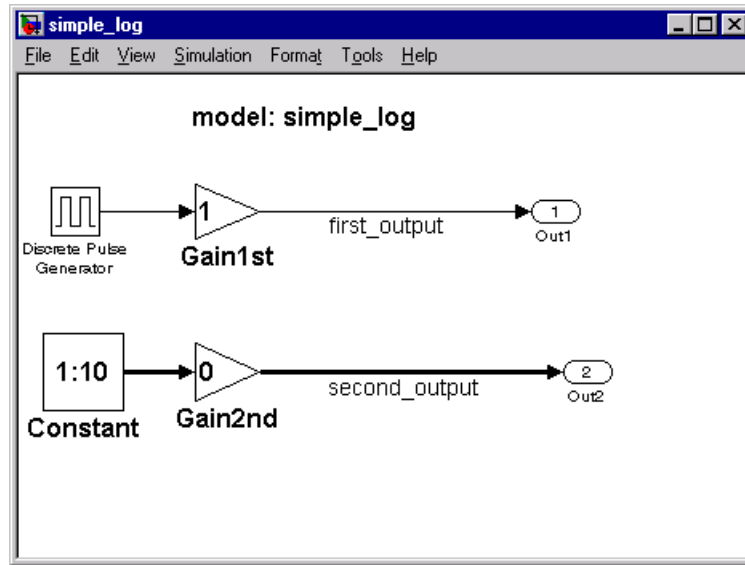
Note For TLC to use this local modified version of `gain.tlc`, which is not the one used by the actual Real-Time Workshop code-generation process, you must make sure the TLC file is in the same directory as the model.

For the purpose of this tutorial, the TLC file contains a bug, so this function cannot be used in an actual scenario.

Tutorial

Generate the Results

- 1 Make sure that you are in the proper directory as given above. This tutorial uses the `simple_log` model, which contains a `Gain` block. Note that the `gain.tlc` used here will be the one in the current directory as the model. This is the `simple_log` model.

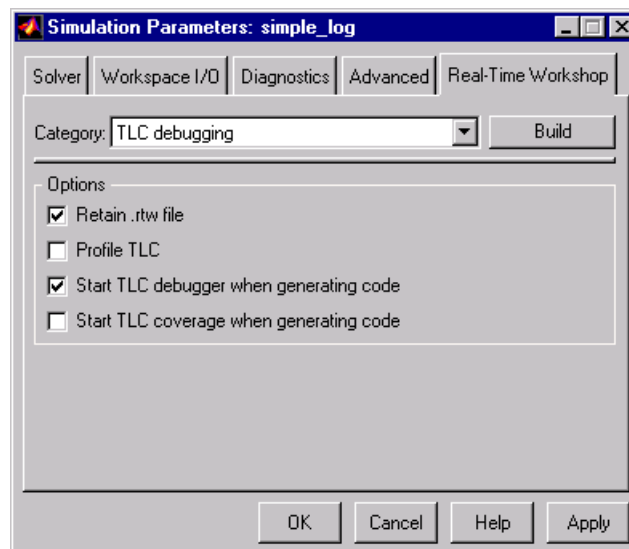


- 2 Simulate the model. This model saves `tout` and `yout` to the workspace, which you can view by selecting **View -> Workspace** from the MATLAB desktop menu.
- 3 Build the model and run the executable. This creates the file `simple_log.mat`. (For more information about the build process, see the *Real-Time Workshop User's Guide*.)
- 4 To compare the results from Simulink and Real-Time Workshop, load the `simple_log.mat` file. The result from Simulink, `yout`, does not match the Real-Time Workshop result, `rt_yout`, because the TLC code contains a bug. You must debug the TLC code to identify the problem.

Invoke the Debugger

- 1 To debug the code, bring up the **RTW options** page by selecting **Tools -> Real-Time Workshop -> Options** from the model's menu. This brings up the **Real-Time Workshop** page of the **Simulation Parameters** dialog box.

- 2 Select **TLC debugging** from the **Category** pulldown. From this screen you can select options that are specific to TLC debugging.
- 3 Select **Retain .rtw file**, otherwise it gets deleted after code generation. Also, select **Start TLC debugger when generating code** to invoke the TLC debugger when starting the code generation process. This is equivalent to adding -dc to the **System target file** field in the **Real-Time Workshop** page of the **Simulation Parameters** dialog box. The dialog box should look like this.



- 4 Apply your changes and press **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 RTW file, and explore variables in your TLC file using `print`, `whi ch`, 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.) Use the

following command and replace *matlabroot* with the full pathname to the location where MATLAB is installed on your system.

```
tlc -dc -r ./simple_log_grt_rtw/simple_log_rtw 'matlabroot/rtw/c/  
grt/grt.tlc' -0. ./simple_log_grt_rtw '-I matlabroot/rtw/c/grt'  
'-I matlabroot/rtw/c/tlc'
```

TLC Debugger Commands

The TLC debugger supports the following commands. You can abbreviate any TLC debugger command to its shortest unique form (use `help command` from within the TLC debugger for more info).

Table 6-1: TLC Debugger Commands

Command	Description
<code>assign variable=value</code>	Change a variable in the running program.
<code>break <"filename":line error warning trace></code>	Set a breakpoint.
<code>clear breakpoint#</code>	Remove a breakpoint.
<code>continue</code>	Continue from a breakpoint.
<code>down [n]</code>	Move down the stack.
<code>help [command]</code>	Obtain help for a command.
<code>list start[,end]</code>	List lines from the file from start to end.
<code>next</code>	Single step without going into functions.
<code>print expression</code>	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> .
<code>quit</code>	Quit the TLC debugger.
<code>status</code>	Display a list of active breakpoints.
<code>step</code>	Step into.
<code>stop <"filename":line error warning trace></code>	Set a breakpoint (same as break).
<code>up [n]</code>	Move up the stack.
<code>where</code>	Show the currently active execution chains.
<code>which name</code>	Looks up the name and displays what scope it comes from.
<code>whos [:: expression]</code>	Lists the variables in the given scope.

To view the complete list of TLC debugger commands, type `help` at the `TLC-DEBUG>` prompt.

Usage Notes

When using `break` or `stop`, use:

- - 1 to break on error
- - 2 to break on warning
- - 3 to break on trace

For example, if you need to break in `gain.tlc` on error, use

```
TLC-DEBUG> break "gain.tlc": - 1
```

When using `clear`, get the status of breakpoints using `status` and `clear` specific breakpoints. For example,

```
TLC-DEBUG> break "gain.tlc": 46
TLC-DEBUG> break "gain.tlc": 25
TLC-DEBUG> status
Breakpoints:
[0] break File: gain.tlc Line: 46
[1] break File: gain.tlc Line: 25
TLC-DEBUG> clear 1
```

In this example, `clear 1` clears the second breakpoint.

Example TLC Debugging Session

When you press the **Build** button and invoke the TLC debugger, you see the TLC-DEBUG> prompt. Note that this example will use the local version of gain.tlc. The simple_log model is configured with the Inline parameters option turned on.

- 1 Place a breakpoint in line 50 of gain.tlc and then continue. To do this, use

```
TLC-DEBUG> break "gain.tlc": 50
TLC-DEBUG> cont
```

This continues the TLC code generation process and invokes the debugger when it reaches the specified breakpoint.

- 2 When control is transferred back to the debugger, use where to show the TLC stack.

```
TLC-DEBUG> where
==> [00] gain.tlc: Outputs(50)
      [01] /devel/R12/.snapshot/nightly.0/perfect/rtw/c/tlc/
commonbodlib.tlc: FcnGenerateConstantTID(407)
```

You can use whos to see the variables in the LOCAL scope and then probe their values.

- 3 If you look at the gain.tlc file, you see the FcnEliminateUnnecessaryParams function has to return the required output code to be dumped in the Outputs function of the C file. You can also look at this using print, and then you can step into the function. For example,

```
TLC-DEBUG> print FcnEliminateUnnecessaryParams(y, u, k)
/* y0[i1] = u0[i1] * (0.0); */
TLC-DEBUG> step
Stopped in file "gain.tlc" at line 23
00023:    %if LiIsEqual(k, "(0.0)")
```

- 4 You probably cannot interpret too much from this, so continue to the next iteration.

```
TLC-DEBUG> cont
TLC-DEBUG> print k
```

```
(1.0)
TLC-DEBUG> print FcnEliminateUnnecessaryParams(y, u, k)
rtb_temp2 = (1.0);
```

- 5** Step into the function `FcnEliminateUnnecessaryParams` and examine outputs in the function.

```
TLC-DEBUG> step
Stopped in file "gain.tlc" at line 23
00023:    %if LibIsEqual(k, "(0.0)")
```

- 6** You are now in the function scope. You do not want to step through the lines in `LibIsEqual`, so use `next` to go to the next line.

```
TLC-DEBUG> next
Stopped in file "gain.tlc" at line 27
00027:    %elseif LibIsEqual(k, "(1.0)")
TLC-DEBUG> print u
rtb_temp2
TLC-DEBUG> print k
(1.0)
TLC-DEBUG> print y
rtb_temp2
```

Looking at this output, you can see that the variable `k`, instead of the input `u`, is assigned to the output `y`. This is done with the statement

```
%<y> = %<k>;
```

This assignment represents the “bug” in this example.

- 7** You can use `up` to go to the `Outputs` function and then down to return to the function scope.
- 8** To fix the bug, you must change `k` to `u`. Try making the change and rebuilding the model. Your results from the simulation and RTW build should match.

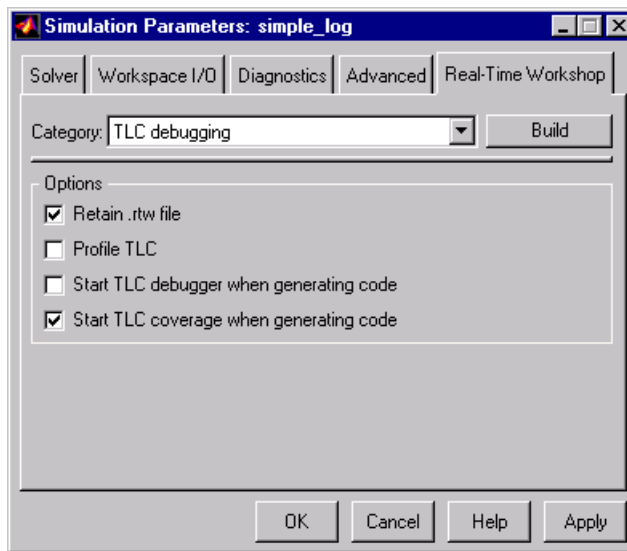
This example covered several ways to use some of the debugger commands. You should try using other commands with this example to familiarize yourself with the various commands.

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 debugging** category of the **Real-Time Workshop** page in the **Simulation Parameters** dialog box. You can also initiate TLC coverage from the command line switch - dg in the **System target file** field, but this is not recommended.



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. The .log file contains usage (count) information regarding how many times it encounters each line during execution. Each line includes the number of times it is encountered followed by a colon and followed by the code.

Example .log File

The .log file for gain.tlc is gain.log. It is shown here when used with the simple_log test model in *matlabroot/toolbox/rtw/rtwdemos/tlctutorial/tlcdebug/*. This file is located in *work_directory/simple_log_grt_rtw/gain.log*.

```

1: %% SRCSfile: gain.tlc, v $
1: %% $Revision$
1: %% $Date: 2000/03/02 22:38:44 $
1: %%
1: %% Copyright 1994-2000 The MathWorks, Inc.
1: %%
1: %% Abstract: Gain block target file
1: %% NOTE: This is different from the file used by Real-Time Workshop to
1: %% implement the Simulink built-in Gain block, and is used here
1: %% only for illustrative purpose
1:
1: %implements Gain "C"
1:
1: %% Function: FcnEliminateUnnecessaryParams=====
1: %% Abstract:
1: %% Eliminate unnecessary multiplications for following gain cases when
1: %% in-lining parameters:
1: %% Zero: memset in registration routine zeroes output
1: %% Positive One: assign output equal to input
1: %% Negative One: assign output equal to unary minus of input
1: %%
1: %function FcnEliminateUnnecessaryParams(y, u, k) Output
2: %if LibIsEqual(k, "(0.0)")
1: %if ShowEliminatedStatements == 1
1: /* %<y> = %<u> * %<k>; */
1: %endif
2: %elseif LibIsEqual(k, "(1.0)")
1: %<y> = %<k>;
1: %elseif LibIsEqual(k, "(-1.0)")
0: %<y> = -%<k>;
0: %else
0: %<y> = %<u> * %<k>;
2: %endif
1: %endfunction
1:
1: %% Function: Outputs===== 1: %% Abstract:
1: %% Y = U * K
1: %%
1: %function Outputs(block, system) Output
2: /* %<Type> Block: %<Name> */
2: %warning LOCAL VERSION OF TLC FILE
2: %assign rollVars = ["U", "Y", "P"]
2:
2: %roll sigIdx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
2: %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)

```



```

2:      %assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
2:      %assign k = LibBlockParameter(Gain, "", lcv, sigIdx)
2:      %if InlineParameters == 1
2:          %<FcnEliminateUnnecessaryParams(y, u, k)>\
2:      %else
0:          %<y> = %<u> * %<k>;
2:      %endif
2:      %endroll
2:
1: %endfunction
1:
1: %% [EOF] gain.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 `gain.log` file, you can see that the code has not been tested with `InlineParameters` off and some cases with `InlineParameters` have not been exercised. Using this log as a reference and creating models to exercise unexecuted lines, you can make sure that your code is more robust.

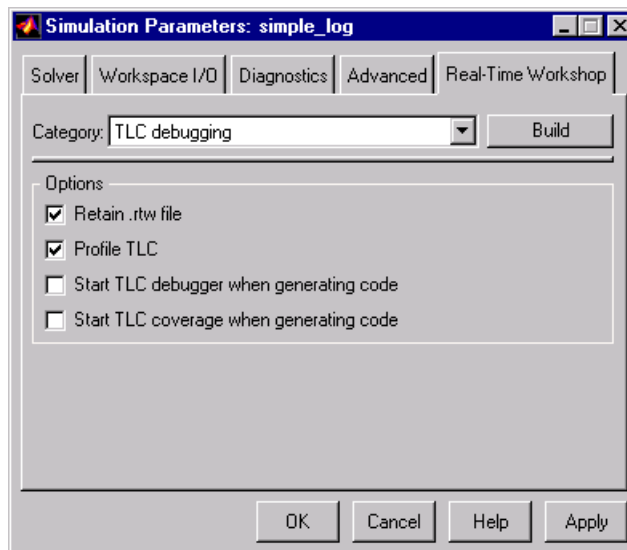
TLC Profiler

This section discusses how to use the TLC profiler to analyze your TLC code to improve code generation performance.

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** page in the **Simulation Parameters** dialog box. Apply your changes and press the **Build** button.



At the end of the build process, the HTML summary and related files are placed in the Real-Time Workshop project directory and the report will be opened in your MATLAB selected Web browser.

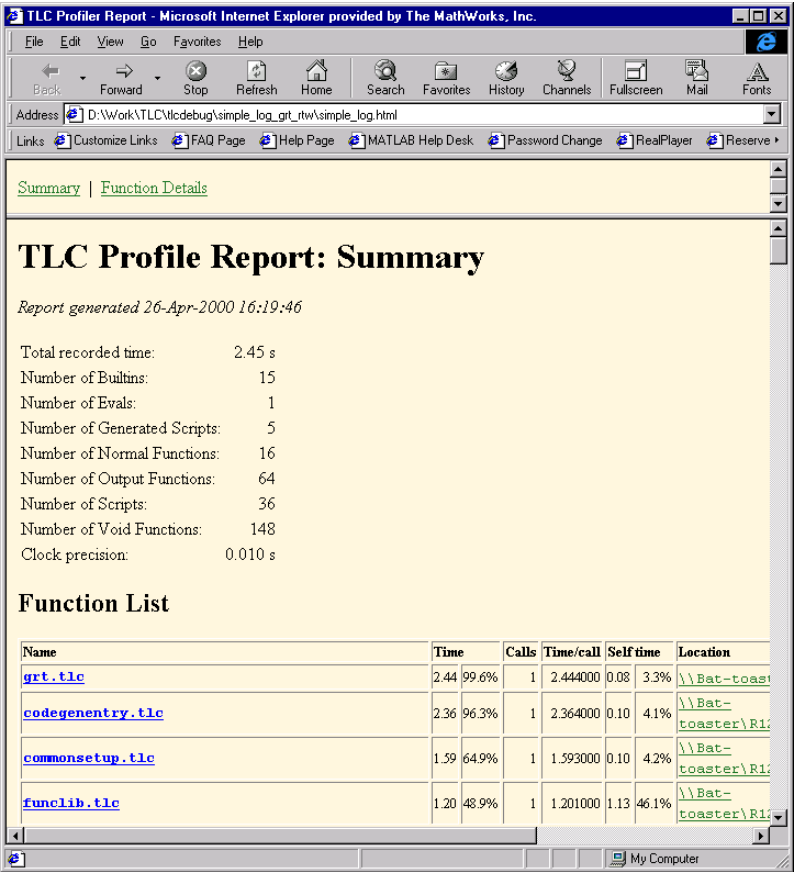
Analyzing the Report

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 calling 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.

This is a portion of the profiler report for the `simple_log` model in `matlabroot/toolbox/rtw/rtwdemos/tlctutorial/tlcdebug`.



The report shows the time spent in the function `FcnEliminateUnnecessaryParams` in `gain.tlc` and other functions, both built-in and library, called during various stages of code generation.

Non-executable Directives

TLC considers the following directives to be non-executable 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 EXIST. Performing an EXIST 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 EXIST 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

Writing Block Target Files to Inline S-Functions	7-2
Fully Inlined S-Functions	7-2
Function-Based or Wrappered Code Generation	7-2
Inlining C MEX S-Functions	7-4
S-Function Parameters	7-5
A Complete Example	7-6
Inlining M-File S-Functions	7-17
Inlining Fortran (FMEX) S-Functions	7-19
Inlining Ada-MEX S-Functions	7-23
TLC Coding Conventions	7-26
Block Target File Methods	7-31
Block Target File Mapping	7-31
Block Functions	7-31
Loop Rolling	7-41
Error Reporting	7-43

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, Ada, or .m, you must inline them in order to have complete code generation for Simulink models that contain them. So once you have determined that you will inline an S-function, you have to 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 `LibAddToModel` Sources for names of the modules containing the supporting functions. This approach is usually done with a one function per file approach 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 will result in 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.

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

The parameters written to the *model.rtw* file are defined in the following table. The `mdlRTWmethod` of level 2 C MEX S-functions is especially useful for adding information to your S-function block records to enable more efficient or elegant inlining. In the following table,

- *Sizes* is an extra parameter of length 2 that gives the number of rows and number of columns in the parameter.
- *Prms* is a type of S-function parameter allowed.

S-Function	No *.tlc	Have *.tlc
Does not have mdl RTW	Sizes => double, Prms => double Error if non-C MEX S-function	Prms => double or string (i nt 16)
Has mdl RTW	Error	No sizes, Prms => any data type

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_sfun.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 User's Guide*.

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)
{

```

```

/* Level 2 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 Mdl Terminate(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);
sfcnTMap[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 Towards 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'ed 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 two fold. 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 mdl Start 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 wish 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 mdl InitializeSizes 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 mdl InitializeSizes, mdl InitializeConditions, mdl CheckParameters, mdl ProcessParameters, and mdl Start functions. In the case of constant signal propagation, mdl Outputs 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 BlockInstanceData and Start methods 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.

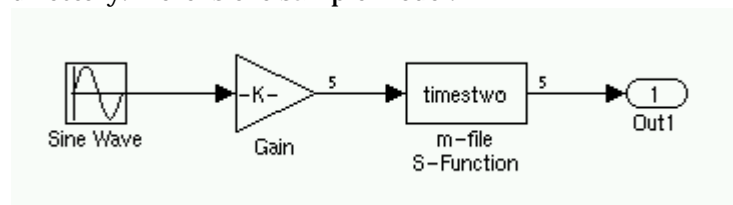
A quick example will illustrate the equivalence of C MEX and M-file S-functions for code generation. The M-file S-function `timestwo.m` is equivalent to the C MEX S-function `timestwo`. In fact, the TLC file for the C MEX S-function `timestwo` will work for the M-file S-function `timestwo.m` as well!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

```

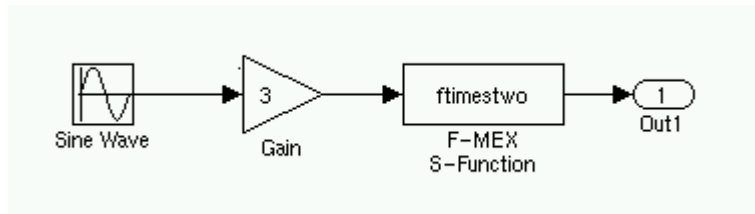
Copy the above code into file `ftimestwo.` for in a convenient working directory.

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 `ftimestwo.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

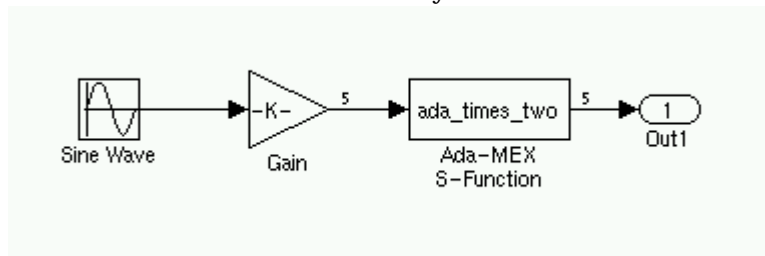
```
/* S-Function Block: <Root>/F-MEX S-Function */
/* Multiply input by two */
rtB.F_MEX_S_Function = rtB.Gain * 2.0;
```

Inlining Ada-MEX S-Functions

Like all the other S-functions, Ada MEX S-functions directly support inlining in the generated code. As an example, copy the `times_two` Ada MEX S-function source code from directory `matlabroot/simulink/ada/examples/times_two/` to a temporary directory — make sure to get both the `times_two.adb` and `times_two.ads` files. Once your Ada MEX environment is set up, compile `times_two` from the MATLAB command line with

```
mex -ada times_two.adb
```

which makes a MEX file named `ada_times_two.mexext`. Use this MEX S-function in a model the same way as other S-functions.



To inline this function in C code, copy the `times_two.tlc` file from the previous M-file S-function example into the temporary directory and name it `ada_times_two.tlc`. In the editor, modify the `%implements` line to match the name of the S-function, in this case it should read

```
%implements "ada_times_two" "C"
```

After ensuring your model is set to use a fixed-step solver, generate C code for this model. The Ada Mex S-function code looks the same as it does for all the other S-function implementations

```

/* S-Function Block: <Root>/Ada-MEX S-Function */
/* Multiply input by two */
{
    int_T i1;
    const real_T *u0 = &rtb_temp1[0];
    real_T *y0 = &rtb_temp1[0];

    for (i1=0; i1 < 5; i1++) {

```

```

        y0[i1] = u0[i1] * 2.0;
    }
}

```

If, as is likely, you wish to generate inlined Ada code for your model and its Ada MEX S-function using the Real-Time Workshop Ada Coder, you use a different TLC file — one that generates Ada code. To get started with this, copy the `times_two.tlc` file from `matlabroot/toolbox/simulink/blocks/tlc_ada/` into the temporary directory and name it `ada_times_two.tlc`. Once again, edit the `%implements` line to read

```
%implements "ada_times_two" "Ada"
```

So the minimum necessary contents of the TLC file look like this.

```

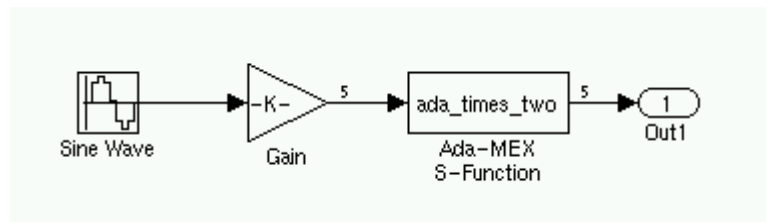
%implements "ada_times_two" "Ada"

%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 select **Ada Simulation Target for GNAT** via the **Tools -> Real-Time Workshop -> Options...** menu using the **Select...** button. This target uses the `rt_ada_sim.tlc` system target file and does not support blocks with continuous states, so it will be necessary to alter the sine wave source block to have a discrete sample time (choose it to be the same as the value chosen for the fixed-step solver's time step, for instance).



Generate code with the **Build** button. The *model*.ads file that is generated will contain this inlined code for the *ada_times_two* S-function.

```
-- S-Function Block: <Root>/Ada-MEX S-Function
-- Multiply input by two
for I1 in 0 .. 4 loop
    RT_B_Temp1(I1) := RT_B_Temp1(I1) * 2.0;
end loop;
```

Notice that the TLC code is relatively independent of the target language in use (C or Ada) — for instance, the `%roll` construct will create the correct C or Ada loop code based solely on whether the block target file implements “C” or “Ada”. TLC constructs themselves are meant to be language-neutral; the only part that is language-specific are the operators, comments, and function name differences in the standard libraries.

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,

```
NumModel Inputs          1
NumModel Outputs         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        "Ti meSt ampA"
        Wi dth      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`, `mdlwi.de.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 `mdlwi.de.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 = %<Li bReal NonFi ni te(i nf)>;
```

Similarly, instead of using %<tTID>, use %<Li bTID()>. For a complete list of functions, see Chapter 9, “TLC Function Library Reference.”

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 = Li bBlockOutputSignal(0, "", lcv, sigIdx)
              %assign u = Li bBlockInputSignal(0, "", lcv, sigIdx)
              %assign k = Li bBlockParameter(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 `xiIdx` 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 xiIdx = [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_
```

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/genmap.tlc`. All the TLC files listed are located in `matlabroot/rtw/c/tlc` for C and `matlabroot/rtw/ada/tlc` for Ada.

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.

```
BlockInstanceData(block, system)
```

```
Enable(block, system)
```

```
Disable(block, system)
```

```
Start(block, system)
```

```
InitializeConditions(block, system)
```

<code>Outputs(block, system)</code>
<code>Update(block, system)</code>
<code>Derivatives(block, system)</code>
<code>Terminate(block, system)</code>

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 the [TLC Function Library Reference](#).

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/tlclookup2d.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)
```

```

%el se
    %assign blockName = LibGetFormattedBlockPath(block)
%endi f
%i f (CodeFormat == "Embedded-C") || (CodeFormat == "Ada")
    %i f !(ParamSettings.ColZeroTechnique == "Normal Interp" && ...
        ParamSettings.RowZeroTechnique == "Normal Interp")
    %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
%endi f
%endi f

```

```
%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 “Chapter 9, “TLC Function Library Reference,”” for a list of relevant functions to call from inside this block function. See look_up.tlc for an example of the BlockTypeSetup function.

Syntax. BlockTypeSetup(block, system) void
 block = Reference to a Simulink block
 system = Reference to a nonvirtual Simulink subsystem

As an example, given the S-function foo 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.

BlockInstanceData(block, system)

The `BlockInstanceData` function is used to allocate persistent data for a block. The code generated from this function is placed in the registration function (*model_reg.h* file) and is called once during model initialization. You should always use this function to allocate data for the following reasons:

- You must never create global variables since they may result in name clashes when you combine models.
- You can statically or dynamically allocate data using this function.
- The Real-Time Workshop declares all its memory in the registration function.

The following example of code determines the method of persistent data allocation based on the `UsingMalloc` flag.

```
%function BlockInstanceData(block, system) Output
%assign pwork = LibBlockPWork(SemID, "", "", 0)
%if EXISTS("ssBlock")
/* %<Type> Block: %<Name> */
```



```

%% Declare the semaphore depending on the Code Format.
/*      VxWorks binary semaphore for task: %<ssBlock.Name> */
%if UsingMalloc == 1
    %<pwork> = malloc(sizeof(SEM_ID));
    rt_VALIDATE_MEMORY(%<tSimStruct>, %<pwork>);
%else
    static SEM_ID %<taskSemaphoreName>;
    %<pwork> = (void *)&%<taskSemaphoreName>;
%endif
%endif
%endfunction

%function Terminate(block, system) Output
%if EXISTS("ssBlock")
%if UsingMalloc == 1
    %assign pwork = LibBlockPWork(SemID, "", "", 0)
    rt_FREE(%<pwork>);
%endif
%endif
%endfunction

```

Note that the use of the macros `rt_VALIDATE_MEMORY` and `rt_FREE` allows this code to work consistently and correctly with all code formats.

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;

%end if
%endfunction

```

Disable(block, system)

Nonvirtual subsystem Disable functions are created whenever a Simulink subsystem contains a block with a Disable function. Including the Disable function in a block's target file places the block's specific disable code into this subsystem Disable function. See `outport.tlc` in `matlabroot/rtw/c/tlc` 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 re-executed 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, ...

```

```

        "<ti magPart>%<i dx>")
    %assi gn pi = Li bBl ockParameter(Value, "", l cv, ...
        "<ti magPart>%<i dx>")
    %<yi> = %<pi>;
    %endi f
%endro ll

%endi f
%endfuncti on %% 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 re-execute 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> */
%<Li bBl ockRWork(PrevT, "", "", 0)> = %<Li bReal NonFi nite(i nf)>;

%endfuncti on %% 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)

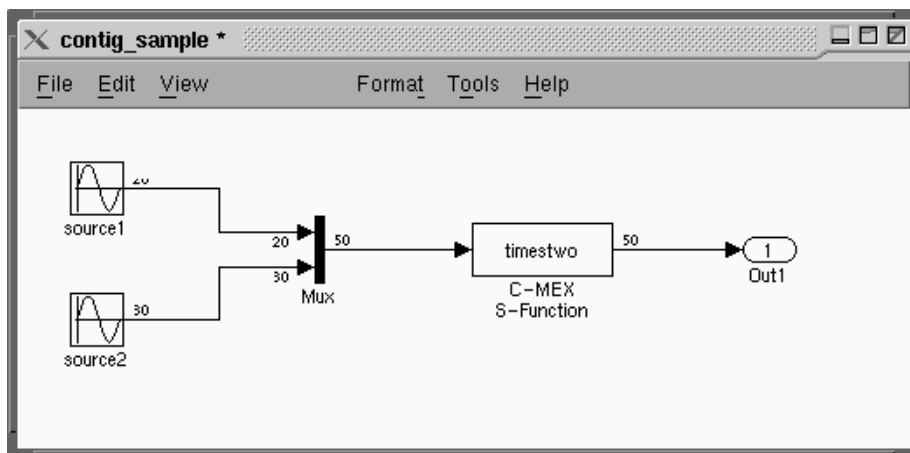
Include a `Terminate` function to place any code into `Mdl Terminate`.

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 {
.
.
  DataInputPort {
.
.
  }
.
.
}
```

From this snippet out 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>/C-MEX S-Function */
/* Multiply input by two */
{
    int_T i1;
    const real_T *u0 = &rtB.source1[0];
    real_T *y0 = &rtB.C_MEX_S_Function[0];

    for (i1=0; i1 < 20; i1++) {
        y0[i1] = u0[i1] * 2.0;
    }
    u0 = &rtB.source2[0];
    y0 = &rtB.C_MEX_S_Function[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, `&rtB.source1[0]`, to the second base address of the signals, `&rtB.source2[0]`. If you do not want to support this in your S-function or your generated code, you can use

```
ssSetInputPortRequiredContinuous(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.

This is accomplished by using the `%roll` directive. There is a tutorial on the `%roll` directive in Chapter 8, “TLC Tutorial.” See also page 5-9 for the reference entry for `%roll` and page 5-29 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 mdl CheckParameters 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`.

TLC Tutorial

Basic Inlined S-Function Written in TLC	8-3
Introduction to TLC Token Expansion	8-6
Building a Model Using the TLC Debugger	8-8
Explore Variable Names and Loop Rolling	8-10
Code Coverage for Debugging TLC Files	8-13
Using a Wrapper S-Function Inlined with TLC	8-15
Inlined S-Function for Dual Port RAM	8-19
“Hello World” Example with model.rtw File	8-23
Generating Auxiliary Files for Batch FTP	8-24
Generating Code for Models with States	8-25
Simulink External Mode and GRT	8-34
Loop Rolling Through a TLC File	8-39

This chapter contains a number of TLC examples. All material needed for the examples, including example models, S-functions, and TLC files is located in *matlabroot/tool box/rtw/rtwdemos/tl ctutori al* . Each example is located in a separate subdirectory. For the duration of this chapter, this directory is referred to as *tl ctutori al* . You may want to copy this directory to a local working area.

This chapter is divided into two sections. The first section is written in a procedural, step-by-step style. The second section is a series of exercises that require you to perform a small amount of research that will result in you becoming more familiar with using this reference guide.

Syntax Highlighting with Emacs

The MathWorks provides Target Language Compiler syntax highlighting for use with Emacs. See Appendix C, “Using TLC with Emacs,” for details on obtaining *tl c-mode* for Emacs.

For those who are new to Emacs, the following provides a quick reference:

Ctrl-x Ctrl-f	Open file into a buffer
Ctrl-x b	Switch to buffer
Ctrl-x 2	Split window
Ctrl-x 1	Expand to one window (i.e., delete other windows)
Ctrl-x Ctrl-c	Exit Emacs
Ctrl-x Ctrl-s	Save buffer
	Standard arrow and delete keys work

Basic Inlined S-Function Written in TLC

Objective: To understand the differences between a noninlined S-function and an inlined S-function. This will help you develop a better understanding of when to use an inlined or noninlined S-function.

Example directory: *tlctutorial/timestwo*

Basic Code Generation

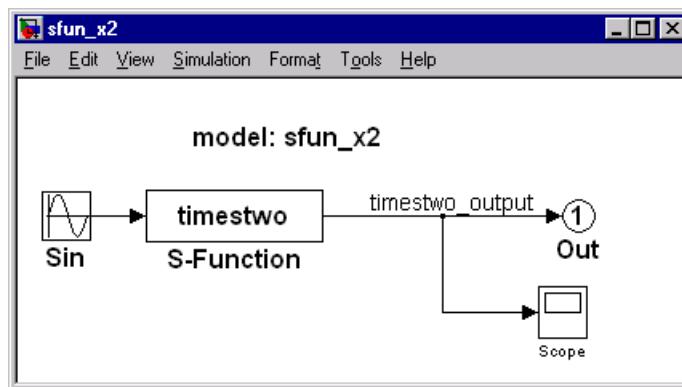
- 1 Copy the Simulink S-function *tlctutorial/timestwo/timestwo.c* into your working directory.

- 2 Create the MEX-file.

```
mex timestwo.c
```

This is needed to avoid picking up the version shipped with Simulink.

- 3 Create the model *sfun_x2* using the Simulink S-function called *timestwo*. Your model should look like this.



- 4 Simulate the model using the fixed-step discrete solver with a step size of 0.01 and stop time of 10.0.

Note that `timestwo.c` code is a level-2 S-function. Level-2 S-functions are beyond the scope of this training session. See the Simulink book *Writing S-Functions* for more information.

Using C coded S-functions with generated code tend to add unnecessary overhead. For simple S-functions where speed is crucial — in cases such as I/O device drivers — you can rewrite the S-function as an inlined S-function by creating a file with the same S-function name with a `.tlc` extension (*filename.tlc*).

- 5 To look at the generated code for the noninlined C coded S-function, select **Generate code only** from the Real-Time Workshop page in the Simulation Parameters dialog and click on the **Generate code** button to generate C code for the model. Now view the Mdl Outputs and Mdl Terminate portions of the generated C code, `sfun_x2.c`. Notice the overhead of calling the S-function. This necessary code allows for a generic API for S-functions. The next step is to inline your S-function by removing the SimStruct associated with your S-function (memory savings) and the generic API (speed up).

Creating an Inlined S-Function

Now, create an *inlined S-function* to replace the `timestwo.c` code. Do this by creating a TLC file named `timestwo.tlc` in your working directory. When the Target Language Compiler detects a TLC file with the same name as the S-function, it uses the TLC version instead of a function call to the external C coded S-function. This process enables TLC to *inline* the code within the generated C code:

- 1 Copy the file `tutorial/timestwo/timestwo.tlc` into your working directory.
- 2 Select **Inline parameters** from the **Advanced** page before you build your executable code. Confirm that your `timestwo.tlc` file works correctly by generating C code, viewing the generated code, running the stand-alone simulation with data logging, and plotting the resulting data.
- 3 Copy the model `tutorial/timestwo/sfun_x2v.mdl` into your working directory. Try using the `timestwo.tlc` file with this model, which is essentially just a vectored version of `sfun_x2.mdl`, and look at the generated C code.

- 4 Close both models (sfun_x2 and sfun_x2v) before continuing with the tutorial.

A key difference when using TLC to inline an S-function is that S-functions often contain empty function calls. Once inlined, the model code only has function calls for functions that actually do something. Empty function calls can be eliminated. Function calls are expensive on DSPs, therefore, eliminating unnecessary function calls can provide better performance, especially when the S-function is used in multiple instances in the same model.

Keep in mind, TLC is *not* a substitute for writing C code S-functions. In order to simulate within Simulink, it is still necessary to have a C code S-function since Simulink does not make use of TLC files. TLC can inline an S-function to make the S-function much more efficient in your RTW target code.

Introduction to TLC Token Expansion

Objective: Achieve a basic understanding of TLC. You will make small textual changes to the generated code to familiarize yourself with TLC token expansion.

The previous example showed the generated code produced by the inlined S-function `timestwo.tlc`. In this example, you will make minor modifications to the `timestwo.tlc` file that will change the generated code:

- 1 Open model `sfun_x2`, which you worked with in the previous example. Generate code using the existing `timestwo.tlc` and review the generated code in the mdl Output section of `sfun_x2.c`.
- 2 Open the `timestwo.tlc` file in the editor of your choice.
- 3 Add any series of C comments to the code in the `Outputs` function in `timestwo.tlc`. When you are done editing, your changes should look something like the following.

```
%function Outputs(block, system)
/* %<Type> Block: %<Name> */
/* C comment #1 */
%%
/* Multiply input by two */
%assign rollVars = ["U", "Y"]
/* C comment #2 */
%assign u = LibBlockInputSignal(0, "", lcv, idx)
%assign y = LibBlockOutputSignal(0, "", lcv, idx)
%<y> = %<u> * 2.0;
%endroll
/* C comment #3 */
%endfunction
```

- 4 Add a TLC comment (or comments) anywhere in the TLC file. TLC comments are defined as text following `%%`. For example,

`%% This is my TLC comment.`
- 5 From the **Real-Time Workshop** page, select the **Generate code only** and generate C code for `sfun_x2`. (Click the **Generate code** button or type ^B.)

View the generated code in the `sfun_x2.c` file and look for the comment lines.

This example should give you insight into how TLC inlining works. You may want to change the algorithm from multiplication to division, or change the 2.0 multiplicand (two times) to some other multiple and regenerate code using the new version of the `.tlc` file. Essentially, you can view TLC inlining as token expansion of the `.tlc` file into the generated code.

Building a Model Using the TLC Debugger

Objective: Introduce the TLC debugger. You will learn how to set breakpoints and familiarize yourself with the various commands supported by the TLC debugger.

Preparing the Example

You invoke the TLC debugger while building the model. Use the same model, `sfun_x2.mdl`, for this example:

- Select output data logging in the model.
- Open the Real-Time Workshop dialog page.
- Change the pull-down menu labeled **Category** from **Target configuration** to **TLC debugging**.
- Select the check boxes **Retain .rtw file** and **Start TLC debugger when generating code**.
- Click the **Build** button.

The MATLAB command window shows the building process information and stops at the `grt.tlc` file and displays the following tag in the command window.

```
TLC-DEBUG>
```

Type `help` to see the list of TLC debugger commands.

Performing the Tasks

- 1 Set a breakpoint on line 19 of `timestwo.tlc`. Type `help break` to list the various syntaxes for the break command. Use the first form of the break command to set a breakpoint on line 19. Type

```
break "timestwo.tlc":19
```

Type `continue` to instruct the TLC debugger to continue to the next breakpoint. The debugger should advance to line 19 in `timestwo.tlc` and display the following status information in the MATLAB command window.

```
Stopped in file "timestwo.tlc" at line 19  
00019: %<y> = %<u> * 2.0;
```

TLC-DEBUG>

- 2 Assign a constant value, e.g., 5.0, to the input signal %<u>.

TLC-DEBUG> assign u = 5.0

- 3 View various entities in the TLC scope. For example,

TLC-DEBUG> print y

TLC-DEBUG> print TYPE(y)

TLC-DEBUG> whos CompiledModel

TLC-DEBUG> print CompiledModel.CodeFormat

- 4 After familiarizing yourself with the viewing capabilities of TLC, type
continue to commence completing code generation.

TLC-DEBUG> continue

Code generation should begin and complete.

- 5 Compare the result with the previous one.

For more information about the TLC debugger, see Chapter 6, “Debugging TLC.”

Explore Variable Names and Loop Rolling

Objective: This example shows how you can influence the generated code from the Simulink GUI.

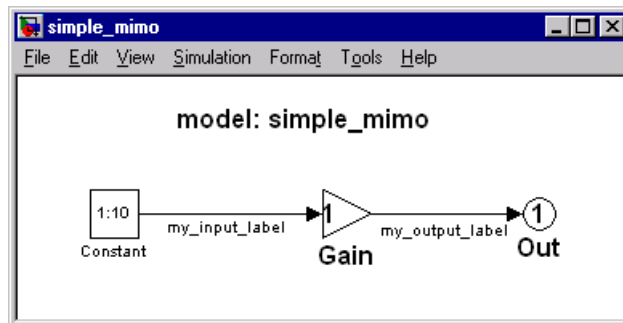
Example directory: *tutorial/mygain*

Preparing the Example

- Start with the model *tutorial/mygain/simple.mdl*.
- Generate the code for model.
- Save *simple* as *simple_mimo.mdl* so that the generated code can be compared easily for both models.

Performing the Tasks

- 1 Change the Sine Wave block to be a Constant block and add labels to the new *simple_mimo* model as shown in this diagram.



- 2 Save your changes in *simple_mimo.mdl*.
- 3 This is a vectorized model. Generate C code for the model and view the Mdl Outputs() section, paying close attention to variables and for loops. This section appears right below the statement

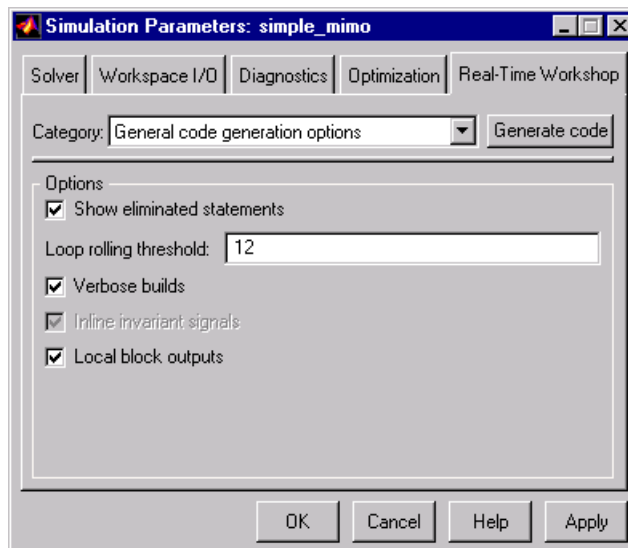
```
/*Gain Block: <Root>/Gain */
```

Notice that:

- The code is rolled into a loop.
 - Pointers *u0 and *y0 are used and initialized to arrays named with the signal names.
- 4 Real-Time Workshop provides some ability for users to control Roll Threshold. You can manually alter the loop rolling threshold by setting your own value for Roll Threshold for the entire model. Do this for your model by selecting **General code generation options** from the **Category** pulldown on the **Real-Time Workshop** page.

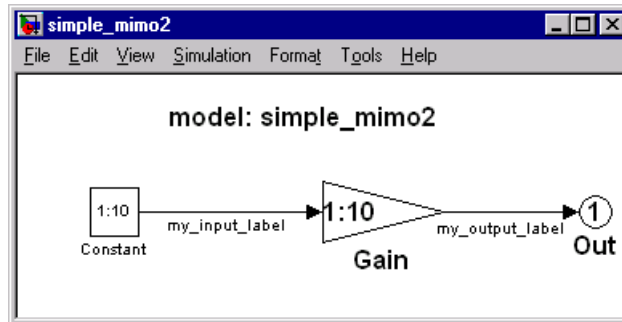
Set **Loop rolling threshold** to 12.

This results in loop rolling only when the number of signals passing through a block exceeds 12.



- 5 Generate code for your model (**Generate code only**) and view the output C file. Note the significant difference in the model code compared to the system that did *not* use such a high Roll Threshold.

- 6 Change the Roll Threshold back to the value 10 in the **General code generation options** dialog. Modify the `simple_mimo` model to have a vectored gain as shown below.



- 7 Once again, generate code and note the difference between this code and the code from the model that simply included a scalar gain of 1. In this case, the 10 gains are placed in a parameter array.

As you look at the TLC file, `matlabroot/rtw/c/tlc/gain.tlc`, notice three "roll Vars" for U, Y, and P. This tells TLC that inputs (U), parameters (P), and outputs (Y) can all be rolled when the number of signals operated on by this block exceeds the Roll Threshold.

Loop rolling is an important but very advanced capability of Real-Time Workshop and TLC. It takes a fair amount of time to fully understand loop rolling before you can apply it. This brief introduction to loop rolling demonstrates that it exists and provides some insight and illustration as to how it works.

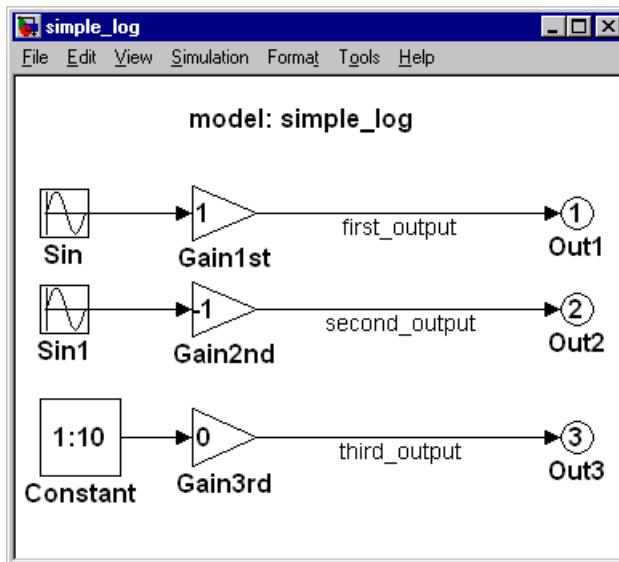
Code Coverage for Debugging TLC Files

Objective: Learn about Real-Time Workshop's capabilities for checking TLC code coverage. This is useful as a debugging tool when writing TLC code. TLC code coverage is easy to use and indicates whether you've exercised all possible cases for generated code that your TLC file can produce.

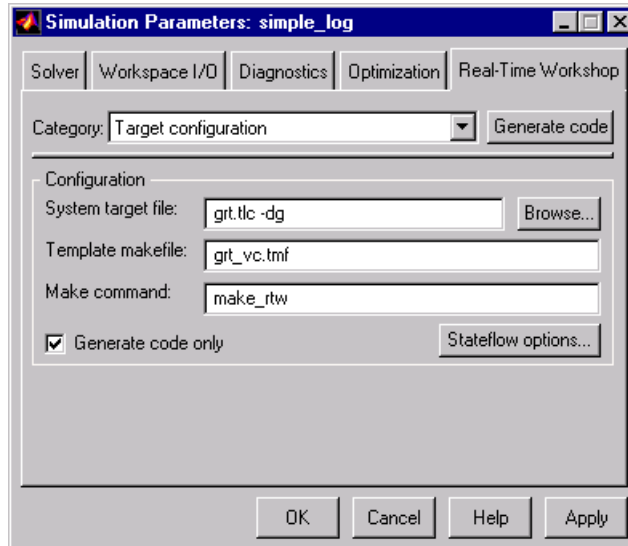
Example directory: *tutorial/codecover*

Performing the Tasks

- 1 Open the model `simple_log`.
- 2 Select the **TLC debugging** from the **Category** pulldown on the **Real-Time Workshop** page and check the **Retain .rtw file** and **Start TLC coverage when generating code** check boxes. This tells Real-Time Workshop to create log files for TLC code coverage. One log file is created for each TLC file used in generating code for the model. This is the block diagram for the `simple_log` model.



- 3 Select **Generate code only**. Be sure **Inline parameters** is *not* selected on the **Advanced** page. Generate code for the model.



Now, look in your build directory and observe the filenames with the `.log` extension (i.e., type: `dir *.log`). These `.log` files correspond to the TLC files with `.tlc` extensions that were used to generate C code for your model.

- 4 Open the log file `gain.log`. Observe the integers 0, 1, 2, ..., n, which indicate how many times the lines of TLC code were exercised. This tool is very useful for debugging custom written TLC files. Aside from TLC files for individual blocks such as `gain.tlc`, TLC relies on additional files such as `blocklib.tlc`, `genmap.tlc`, `mdlhdr.tlc`, `mdlbody.tlc`, `mdlparam.tlc`, `mdlvars.tlc`, etc.
- 5 Select the **Inline parameters** check box and generate code once again. View `gain.log` and `simple_log.c` and pay attention to the model output code for the three outputs. Is it clear how the code has changed and why? The generated code is optimized for speed, however, it is not suited for parameter tuning.
- 6 You may also try playing with the various parameters, such as changing the source block for subsystem 2 from a discrete to a continuous sine wave,

changing Gai n3rd from 0 to 1, and seeing how the .log file and the generated C code change with and without inlined parameters.

Using a Wrapper S-Function Inlined with TLC

When to Consider?

Assume that the file `wrapfcn.c` contains an existing function called `wrapfcn`. In this case, you have the source code, however, making the correct changes for compilation, you could also link against an existing library if the source code is not available. For example, suppose you had an existing object file compiled for a TI DSP, or, for some other processor where Simulink does not run. You could write a dummy C S-function and use a TLC wrapper that would provide a call to the external function, for which you do not have its source code. For instance, this could be the case if you had a library for an FFT algorithm optimized for your target processor. You would also make appropriate changes to a template makefile, or, somehow provide a means of linking against the library. Now that you understand why you might want to do this, let's look at an example.

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

The purpose of a wrapper S-function is to enable a user to pull in an existing C or Ada function without fully rewriting it in the context of a Simulink S-function. Basically, you write a simplified S-function that merely calls the existing, external function.

Caveat Using the object file without the source code only works at present with the Microsoft Visual C/C++ Compiler (MSVC). In this case, the object file must also have been created with the same compiler (i.e., MSVC).

The only requirement for the dummy C coded S-function is to make sure that it uses the correct number of inputs and outputs. Other than that, the S-function does not need to provide correct computations, unless we find it necessary to use these computations to obtain correct simulation results in Simulink – prior to generating real-time code.

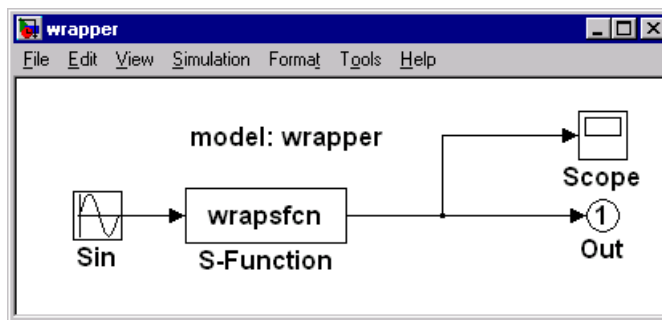
Exercise

The “external function” is provided in the file `wrapfcn.c`. You are also provided with a C code S-function called `wrapsfcn.c`. Now you must create a wrapper TLC file that:

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

The files for this exercise are in the `tutorial/wrapper` directory of the TLC examples directory.

This is the block diagram for the wrapper S-function example.



You can run the C source code to view the correct results.

The TLC Wrapper

TLC is able to handle port inputs and outputs via the following functions.

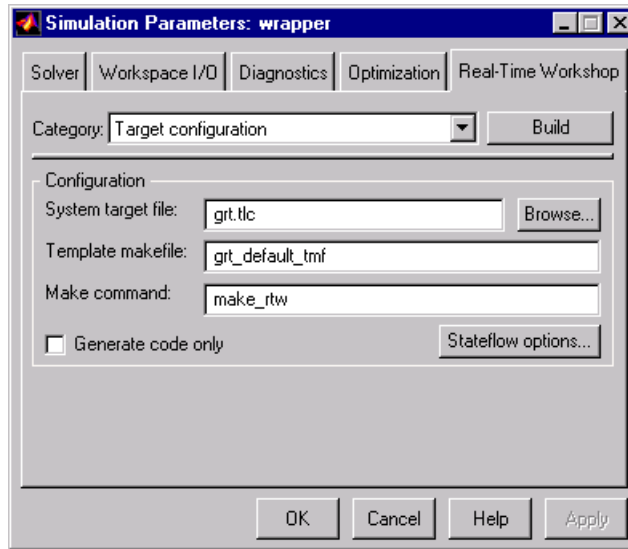
```
%assign u = LibBlockInputSignal(0, "", "", 0)
%assign y = LibBlockOutputSignal(0, "", "", 0)
```

Copy the `wrapsfcn_assign.tlc` file to `wrapsfcn.tlc` and try to create the inlined S-function wrapper.

Note that this also requires you to add the following options to the model's parameters.

```
set_param('wrapper/S-Function', 'SFunctionModules', 'wrapfcn');
```

Use the `SFunctionModules` parameter to inform the RTW build process that it needs to compile and link with `wrapfcn.c`. See the Simulink book *Writing S-Functions* for more information about S-function modules.



Note You should not continue until you have made a reasonable attempt to solve the problem.

Solution

```
%% File      : wrapsfcn.tlc
%% Abstract:
%%          Example tlc file for S-function wrapsfcn.c
%%
%% implements "wrapsfcn" "C"
%%
%% Function: BlockTypeSetup =====
%% Abstract:
%%          Create function prototype in model.h as:
%%          "extern double wrapsfcn(double u);"
%%
%%function BlockTypeSetup(block, system) void
```

```

%openfile buffer
%% Assignment: provide one line of code as a function prototype
%% For "wrapfcn" as described in the assignment.
extern double wrapfcn(double u);

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

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

%endfunction %% Outputs

```

Inlined S-Function for Dual Port RAM

Objective: Learn how you can use an inlined S-function to access hardware with minimal overhead. Also learn how to include parameter values from a Simulink block's mask.

Real-Time Workshop includes runtime interface files for DOS and Tornado/VME targets that are able to access I/O devices, provided you have the additional hardware.

Note The generated code for this example is for illustrative purposes only and the resulting executable file will not work (it will cause an exception violation).

Exercise

The files for this exercise are in the *tutorial*/dual port ram directory of the TLC examples directory.

You are given the C code S-function `dp_read.c`. Write your own TLC file to inline this S-function. The `dp.mdl` model file contains the *dual port read* model. The mask for the `dp_read` S-function allows you to enter parameters for:

Parameter	Description
offset	Scalar offset value
gain	Gain
varType	Data type for the value read from dual port RAM
hwAddress	Memory address that your (fictitious) custom hardware needs to access

The S-function `dp_read.c` is written with error handling to ensure dialog box entries are consistent with the types of data that are to be included in the generated code. This error handling functionality is only needed in the S-function, and is not needed in the generated code.

To simplify your assignment, the file `dp_read_assign.tlc` is included. You must:

- Copy this file to `dp_read.tlc`.
- Insert your own TLC code.
- Test by selecting **Generate code only**.
- Use your editor to view the resulting model code once you have generated the code.

This is the file you will need, `dp_read_assign.tlc`. Copy it to `dp_read.tlc`.

```
%% File      : dp_read.tlc
%% Abstract:
%%      Dual-Port Read block target file for S-function dp_read.
%%
%% Copyright (c) 1994-1999 by The MathWorks, Inc. All Rights Reserved.
%%

implements "dp_read" "C"

%% Function: Outputs =====
%% Abstract:
%%      Dual port read for the dp_read.mex S-function, (e.g. dp_read.dll).
%%      Read the value from the specified address entered in the S-function
%%      parameters and write to the output location:
%%      Y = *dpAddress
%%
%%      In the dp_read, the Parameters are defined to be:
%%      offset, gain, variableType, hardwareAddress
%%
function Outputs(block, system) Output
/* %<Type> Block: %<Identifier> */
{
    %% ONLY MAKE CHANGES TO THE CODE BELOW HERE!
    %% #1 and #2
    %assign offset    = CAST("Real", P1.Value[0])
    %assign gain      = %% COMPLETE THIS LINE - Look at mask for the
                        %% dp_read block and also look at the .rtw file after
                        %% generating it by using >> rtwgen dp
                        %%
    %% #3
    %assign varType   = %% COMPLETE THIS LINE - using LibBlockParameterString
                        %% see page 26 of TLC Quick Ref
    %% #4
    %assign hwAddress = %% COMPLETE THIS LINE - similar to varType, except P4
    %% #5
```

```
%assign y          = %% COMPLETE THIS LINE - using LibBlockOutputSignal
                    %% where portIdx=0, ucv="", lcv="", sigIdx=0
%% However, this model uses the "y" variable for the simple
%% case of only a scalar output. See the LibBlockOutputSignal function
%% on page 20 of the Target Language Quick Reference.

%% The generated code for this block should appear as follows:
%%
%% volatile float *dpAddress = (float *) 0x40;
%% rtB.sl_dual_port_read = ((real_T) *dpAddress) * 1.0 + 0.0;

volatile %<varType> *dpAddress = %% COMPLETE THIS LINE
%<y> = %% COMPLETE THIS LINE

}

%endfunction %% Outputs
```

Further Hints

- In the Outputs section, you will need five lines that start with %assign.
- The first two use CAST to ensure that parameter values entered in the mask as 1 are cast to a real 1.0.
- The CAST function is not required for varType and hwAddress since varType is a string while hwAddress is a hex value that is used with no change.
- For further insight, look at the rtw file called dp.rtw. (You can use rtwgen to generate the rtw file for the model while you are iterating on your own TLC file.

rtwgen dp

- For the third and fourth %assign's for varType and hwAddress see page 8-21 for handling the string that must be read in from the .rtw file. Use a library function. See Chapter 9, "TLC Function Library Reference."
- Use a fifth line starting with %assign to setup the variable y as an output.
- The final two lines in the Outputs function produce the lines of code that appear in the mdl Output section of the generated code.

Note You should not continue until you have made a reasonable attempt to solve the problem.

Solution

```

%% File      : dp_read.tlc
%% Abstract:
%% Dual-Port Read block target file for S-function dp_read.
%%
%% Copyright (c) 1994-1999 by The MathWorks, Inc. All Rights Reserved.
%%

%implements "dp_read" "C"

%% Function: Outputs =====
%% Abstract:
%% Dual port read for the dp_read.mex S-function, (e.g. dp_read.dll).
%% Read the value from the specified address entered in the S-function
%% parameters and write to the output location:
%%      Y = *dpAddress
%%
%% In the dp_read, the Parameters are defined to be:
%%      offset, gain, variableType, hardwareAddress
%%
%function Outputs(block, system) Output
/* %<Type> Block: %<Identifier> */
{
    %assign offset    = CAST("Real", P1.Value[0])
    %assign gain      = CAST("Real", P2.Value[0])
    %assign varType   = Li bBlockParameterString(P3)
    %assign hwAddress = Li bBlockParameterString(P4)
    %assign y         = Li bBlockOutputSignal(0, "", "", 0)
    volatile %<varType> *dpAddress = (%<varType> *) %<hwAddress>;
    %<y> = ((real_T) *dpAddress) * %<gain> + %<offset>;
}

%endfunction %% Outputs
    
```

“Hello World” Example with model.rtw File

Objective: What programming class would be complete without “Hello world”? In this exercise, you’ll learn how to make simple modifications to `grt.tlc` to generate a separate output file containing your code for `hello.c`.

What benefit is there in this assignment? More advanced TLC programming applications may rely on your ability to generate a file. This example prepares you for virtually any auxiliary files.

Exercise

- 1 There is a special generic real time file called `grt_assign.tlc` in `matlabroot/toolbox/rtw/rtwdemos/tutorials/helloworld`. Copy this file to `grt.tlc` and create the necessary code to create `hello.c`. You can run any model, for example, `f14` or `vdp` from Simulink demo and, as long as you use your special version of `grt.tlc`, you will generate the file `hello.c`.
- 2 Edit `grt.tlc` and make changes accordingly.
- 3 Before you try to build, select the **Generate code only** check box on the **Real-Time Workshop** page, which allows you to generate the C code and the makefile.
- 4 Now you can generate code and get the file `hello.c`.

Generating Auxiliary Files for Batch FTP

Objective: One application that could use an auxiliary file is automating the process of downloading source files (e.g., *model.c*, *model.h*, *model.rtw*, *model_prm.h*, *model_reg.h*, *model.mk*) to an external target. Such cases arise when the compiler resides on the machine. For example, LynxOS and QNX are UNIX-based systems that allow you to run a compiler under the RTOS where the real-time executable will run.

The files for this exercise are in the *tutorial/addinfo* directory of the TLC examples directory.

Exercise

- 1 Copy the file *grt.tlc* (in *matlabroot/rtw/c/grt/grt.tlc*) to your current working directory.
- 2 Add one line to include *ftpdownload.tlc* in *grt.tlc*.
- 3 Copy the *ftp_assign.tlc* file to *ftpdownload.tlc* into the same directory.

This exercise only requires you to add the ability for TLC to extract the model name and place it in the file that is generated. The information required can be found in the first few lines of the *model.rtw* file.

- 4 Modify the *ftpdownload.tlc* file so that it extracts the model name and outputs the appropriate filenames to a file called *download* along with the *ftp* commands.
- 5 Open a model (any of the *.mdl* files used thus far) and generate code for it using the modified *grt.tlc* file that you have just created. Look at the file *download* to see if everything has proceeded okay.

Generating Code for Models with States

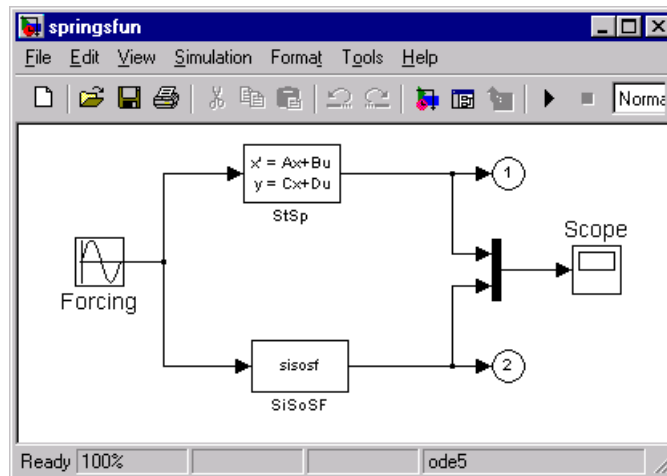
Objective: Look at Real-Time Workshop code generation for the models that have states. This does not involve many new Real-Time Workshop concepts, but it is instructive to see the changes in the generated C code as opposed to the `sfun_x2.mdl` on page 8-3, for example.

This is not an assignment per se, but rather a demonstration of the process involved, from designing a model to generating and using the code. This choice of model involves some subtleties, which you will see in the discussion ahead:

- 1 Open the `springsfun.mdl` model located in the `tutorial/states` subdirectory. This is the model of a linear system with transfer function

$$TF(s) = \frac{a_0}{S^n + a_{n-1}S^{n-1} + \dots + a_1S + a_0}$$

that represents a simple **SISO** system. The picture of the model is given below.



- 2 Before executing the model, run the m-file `initialize.m` that is in the same subdirectory. This generates initial parameters for the system. These parameters are in the form of a random, stable, eighth order linear system. Once this is done, the model is ready to execute.

- 3 Examine the properties of the SiSoSF S-function block and the state space block. Both of these blocks are intended to do exactly the same thing, so that they can be checked against each other. The S-function, `si sosf`, is the state space equivalent of *TF* above, in the controllable canonical form. Therefore, the system is described by x_0 being the state in question

$$\dot{x}_0 = x_1, \dot{x}_1 = x_2, \dots, \dot{x}_6 = x_7$$

$$\dot{x}_7 = - \sum_{i=0}^7 a_i x_i + u(t)$$

where $u(t)$ is the control signal. The output $y(t)$ is given by $y(t) = a_0 x_0$.

This same information is incorporated in the state space block, via matrices *a*, *b*, *c*, and *d*. The order of the states, however, may be different from that given above, with the differences being those of nomenclature, and not of fact.

- 4 The S-function is driven by the program `si sosf. c`, which is also found in this directory. If the precompiled version of `si sosf. c` (`si sosf. dll`) is not in your directory, you will need to compile it with the command

```
mex si sosf. c.
```

Also, study the `si sosf. c` program. This program was created by editing a copy of the `sfuntmpl. c` program template, which is what you are advised to do. Note the various macros (`ssNumSetSFcnParams`, `ssSetSFcnParamNotTunabl e`, `ssGetContStates`, etc.) that are used to access the `Si mStruct` structure to get/set the various model parameters. These macros are defined in `si mstruc. h` and described in the Simulink book *Writing S-Functions*. Some things to be noted are:

- The parameter vector that is passed to Si SoSF are the a_i values, which form the last line of the *A* matrix. The length of this vector determines the number of states (`Mdl I ni ti a l i zeSi zes`).
- The initial conditions have been chosen as $x_0 = 1.0$, $x_i = 0.0$ if $i \neq 0$. This is an arbitrary choice, and a real program (as opposed to an example) would

have this as a parameter. However, this choice has been made to keep things simple (MdlInitializeConditions, which is optional).

- The MdlDerivatives function is the heart of the program and should be studied carefully. The xdot vector stores the derivatives, and xdot[7] (in this case) and more generally xdot[NCStates-1] is calculated incrementally. Additionally, the fact that x and xdot have disparate indices in the line

```
xdot[i] = x[i+1];
```

should be noted, since this causes some complications in the TLC file.

- 5 The next step is to write a .tlc file to enable the inlining S-function. This has already been done for you, and the file is called sisosf.tlc. A listing of the file follows.

```
%% FileName: sisosf.tlc
%%
%% Purpose: To implement a SISO system in controllable canonical form.
%% implements "sisosf" "C"

%% Function: Outputs =====
%%
%function Outputs(block,system) Output
%assign y = LibBlockOutputSignal(0,"", "", 0)
%assign x = LibContinuousState("", "", 0)
%assign dc = LibBlockParameter(P1,"", "", 0)
%% Dc gain is the negative of the first element of the

%% The first state (zeroth derivative) is the output
/* Block %<Name>: %<Type> Output */
%<y> = -%<dc> * %<x>;
%endfunction %% Outputs

%% Function: Derivatives =====
%%
%function Derivatives(block,system) Output
%assign ncStates = ContStates[0]
%assign offset = ContStates[1]
%assign input = LibBlockInputSignal(0,"", "", 0)
/* Derivatives function starts here */
{
    real_T *dx = ssGetdX(%<tSimStruct>);

    dx[%<ncStates> - 1 + offset] = %<input> + \
                                   %<LibBlockParameter(P1,"", "", 0)> * \
                                   %<LibContinuousState("", "", 0)>;
%assign rollVars = ["Xc", " <param>/P1"]
```

```

%roll xIdx = [0: %<ncStates-2>], xlcV = RollThreshold, block, "Roller", rollVars
%assign idx = (xlcV != "") ? "%<xIdx+offset> + %<xlcV>" : "%<xIdx+offset>"
%assign lcv = (xlcV != "") ? "1 + %<xlcV>" : ""
%assign x = LibContinuousState("", lcv, xIdx+1)
%assign a = LibBlockParameter(P1, "", lcv, xIdx+1)
/* Block %<Name>: %<Type> derivatives */
dx[%<i dx>] = %<x>;
dx[%<ncStates - 1 + offset>] += %<a> * %<x>;
%endroll
}
%endfunction    %% Derivatives

%% Function: InitializeConditions =====
%%
%function InitializeConditions(block, system) Output
%assign ncStates = ContStates[0]

%% Initialize State zero (position) to 1.0, rest are 0.0
%assign x = LibContinuousState("", "", 0)
/* %<Type>: %<Name> Position Initialization */
%<x> = 1.0;

%foreach sigIdx = ncStates - 1
    %assign x = LibContinuousState("", "", sigIdx+1)
    /* %<Type>: %<Name> Other States Initialization */
    %<x> = 0.0;
%endforeach
%endfunction    %% InitializeConditions

%% [EOF] sisosf.tlc

```

Derivatives Function

The `Outputs` and `InitializeConditions` functions in this TLC file roughly parallel the corresponding functions in the `sisosf.c` program and thus will not be elaborated on. The heart of the program is again the `Derivatives` function, where the derivatives of the states (assigned to the `dx` vector) will be computed. Since the expression for the derivatives for all but the last state is similar, it is put in a loop. However, the discrepancy in the indices of `dx` and `x` must be treated with care.

The main section of the `Derivatives` function appears inside the `%roll...%endroll` construct. The utility of this method as opposed to, say, `foreach`, is that if there are only a few states (less than `RollThreshold`, which is set to 5 by default), the expressions are explicitly written out, since using a `for` loop would cause unnecessary overhead. However, when there are more states (especially when there are very many), the code is put in a `for` loop. The

`%roll` command is explained in the “Compiler Directives” section of this document. However, this section points out some features as well.

The states run from 0 to $(ncStates - 1)$, and therefore, so do the derivatives. However, when the `dx` vector is returned by the call to the `ssGetdX` macro, the pointer for the entire set of derivatives is returned, not just the ones corresponding to this block. In fact, this model has been designed so that the derivatives from the `dx` block appear *before* those from the `Si SoSF` block in the generated code. Therefore, `dx` corresponding to x_0 is actually `dx[8]` in this example. For this purpose, we use the statement

```
%assign offset = ContStates[1]
```

in the beginning of the function. Alternatively, `dx` could have been declared as

```
real_T *dx = ssGetdX(%<tSimStruct>) + offset;
```

and then the `offset` term could have been dropped, since `dx[0]` would correspond to x_0 .

Next, since `dx[ncStates - 1]` needs special treatment, the looping is done over the range $[0 : ncStates - 2]$. This appears in the `%roll` statement, reproduced here for convenience

```
%roll xIdx = [0:%ncStates-2], xlcV = RollThreshold, block, "Roller", rollVars
```

Parameters

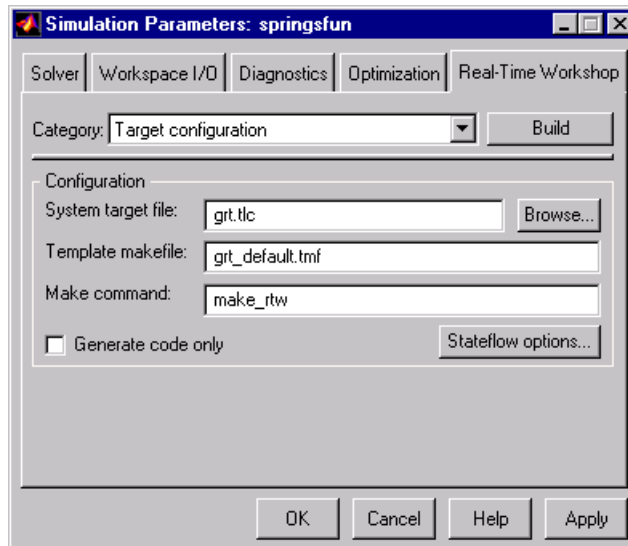
The parameters are as follows: `xIdx` plays a dual role. If the loop does roll, i.e., a `for` loop is generated, then `xIdx` is set to the first value of the `RollRegon` (in this case 0). The `for` loop always starts from zero and goes up to one less than the width of the `RollRegon`. The `RollRegon`, which usually (but not in this case) is set to the eponymous identifier from the *model.rtw* file, contains the vector(s) upon which the TLC compiler bases its decision whether or not to roll.

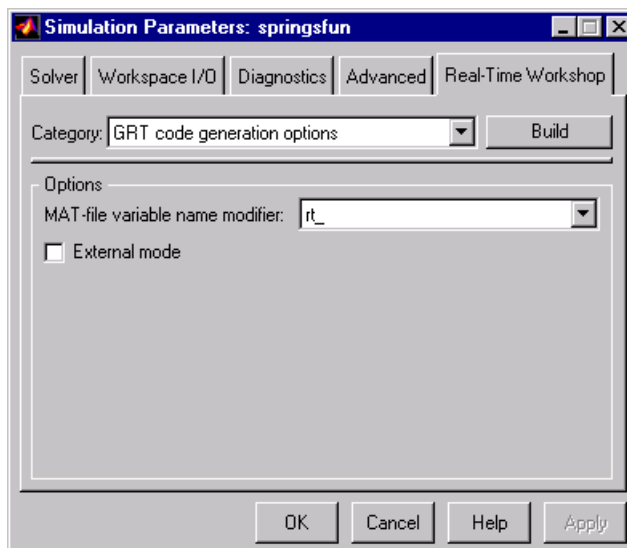
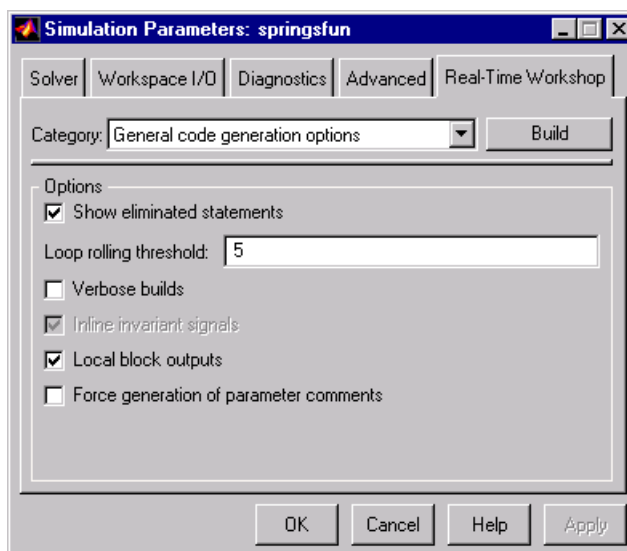
In this case, we have a customized vector, i.e., $[0 : ncStates - 2]$ as our `RollRegon`, and the `xlcV` variable is set to the (string) value of the loop index. The parameter `rollVars` specifies the variables over which the rolling takes place. If rolling does occur, a vector that starts from the appropriate place in (say) the state vector will be declared; similarly for the parameter vector. The “appropriate place” will be given by the value of `xIdx`.

If the loop does not roll, `xIdx` goes from 0 to one less than the width of the `RollRegon`, irrespective of the actual start and end points of the vector. In this mode, the `%roll` command behaves identically to the `%foreach` command.

However, the Roll Vars are appropriately offset so that the correct elements of the state and parameter vectors (in this case) are used. This is given by the first element of the Roll Region.

As an exercise, generate the code using the **Build** command in the **Simulation Parameters** dialog box. Check that the **Loop rolling threshold** option is set to 5 (less than the width of the Roll Region) in the **Options** section of the dialog box.





Study the following lines of code from the `Derivatives` function of the `sisosf.tlc` file to see how they change. This gives important insights into the functionality of `%roll`.

```
/* offset = %<offset> ncStates = %<ncStates> */
/* xIdx = %<xIdx> xlcV = %<xlcV> lcv = %<lcv>*/
```

Now, set the **Loop rolling threshold** to 10 so that the code **does not** roll. Study the code again, and see how the various parameters have changed.

Additional Exercise

- 1 Copy `gain.tlc` from `matlabroot/rtw/c/tlc` to your working directory.
- 2 Change `RollRegions` in the lines below to arbitrary vectors, e.g., `[0: 10, 11, 12, 13: 19]`.
- 3 Now use the `simple_mimo2.mdl` model from an earlier assignment, only changing the inputs to `[1: 20]` (to match our `RollRegion`), and the gain to `[20: -1: 1]` to match the inputs.
- 4 Set the **Loop rolling threshold** to 12 (no roll), 8 (second region rolls, first does not) and 5 (neither region rolls). The relevant section of `gain.tlc` is given below.
- 5 Add comment lines as above to see what the values of `sigIdx`, `lcv`, etc. become.
- 6 Change `RollRegions` below to `[0: 10, 11, 12, 13: 19]`

```
%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)

<snip>

←Insert comment lines here before %endroll
%endroll
```

This is not, of course, a recommended practice for writing TLC files (i.e., hard coding roll regions), but is merely an exercise to see how `%roll` works. You may also try changing the `RollRegion` to `[3: 10, . . .]` etc., and `sigIdx` to `sigIdx-1` in the `%assign u` line.

When you are done, do not forget to change the name of `gain.tlc` to `gain_old.tlc` so that future code generation endeavors involving the Gain block are not hampered.

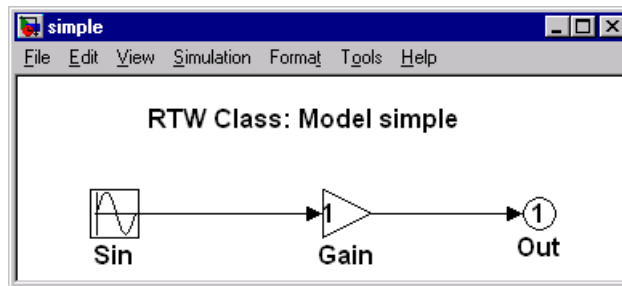
Simulink External Mode and GRT

Objective: Become familiar with GRT, Simulink External Mode, and data logging. Before you begin, be sure to switch into an empty work directory.

Running External Mode in Simulink

- 1 Create the model `simple.mdl` consisting of
 - sine wave source block (Sources)
 - unity gain block (Linear)
 - outputport (Connections)

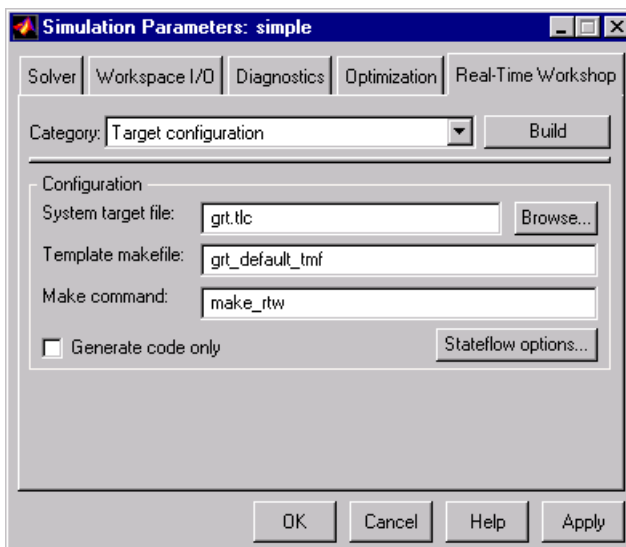
It should appear as the following.



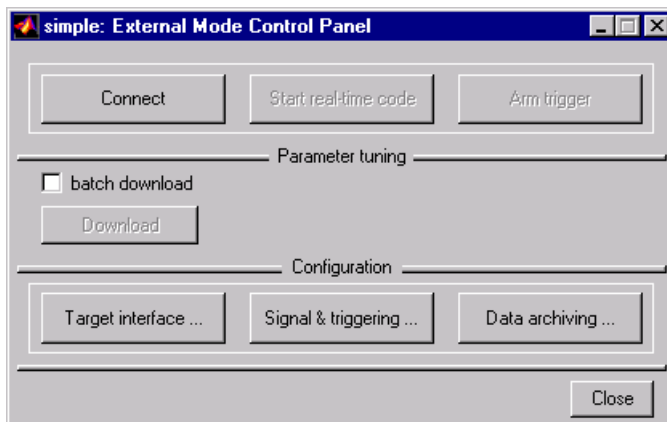
- 2 Set the following dialog options (**Tools -> Real-Time Workshop -> Options**)
 - Solver page
fixed-step, discrete, step-size = 0.01 seconds, Stop time = 1000.0
 - Workspace I/O
save Time as `tout`, save Output as `yout`, limit to last 1,000 pts., decimation = 1 (e.g., no decimation)
 - Real-Time Workshop page
System target file: `grt.tlc`
Template makefile: `grt_default.tmf` (optionally: `grt_watc.tmf`, etc.)
Make command: `make_rtw`
Retain .rtw file

- Save simple.mdl (**File -> Save As**)

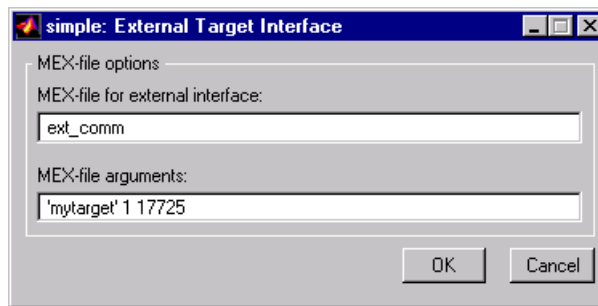
Observe the dialog box entries in the following figure.



- 3 Open **External Mode Control Panel** from the **Tools** menu.



- 4 Open the **Target interface ...** dialog and specify the MEX-file for external interface. For MEX-File arguments, the figure below options for the node name, verbosity = 1, and a port address that is selected as 17725. Defaults for MEX-file options would be your local host name and port 17725. Therefore, in this particular example, they could have been omitted. For targets where the target has a separate network ID, appropriate entries would be required.



- 5 From the **Real-Time Workshop** page, go to the **Options...** dialog to select **External mode** check box. Click **Build** to generate the code and create the executable for non-real-time execution. Note that in order for this exercise to work, you *must* run the model for enough time steps so that you have enough time to change parameters before the model completes and ends. This explains the high value of stop time (1000.0 seconds) chosen earlier. You can also specify the running time when you execute the code.
- 6 Open a DOS prompt window (i.e., a virtual DOS machine) and run the non-real-time executable. (Type `simple` at the DOS command prompt.)

There are some options available for you to choose the running type. (Type `simple -tf` for details.)

- `-tf 20` Sets final time to 20 seconds, `inf` runs forever.
- `-port 300` Sets port to 300.
- `-w` Waits for start message from host.

- 7 While the stand-alone simulation is running, start Simulink external mode.
 - Select **External** from the **Simulation** menu.
 - Go to **External Mode Control Panel** and click **Connect** to start.

- Immediately open the Gain block and change its value.

Once the execution is completed, it should create an output file, `simple.mat`.

- 8 To load the generated MAT-file into MATLAB and plot the results, use

```
load simple
whos
plot(rt_tout, rt_yout)
```

Advanced External Mode Exercise

Note This section is entirely optional.

Starting with the model `simple`, add a Constant block (value of constant set to zero). Add a Stop Simulation block (Sinks) downstream from the Constant block. Save this model as `simple_s.mdl`.

- 1 Set the Stop Time to 200,000. From the **Code Generation Options** dialog box, select **External mode** and **Verbose builds**. Save the model as `simple_s.mdl`. Generate and build the executable. Be sure to select **limit rows to last 1,000**. (Workspace I/O page)
- 2 Before running the executable, look at the **Simulation** menu items that specify external mode. You'll need to observe the following sequence:
 - Start the executable running, i.e., type `simple_s` in the DOS window.
 - Select Simulink external mode by selecting **Simulation -> External**.
 - Select **Connect to target** from the **Simulation** menu or from **External Mode Control Panel**.
 - Change the gain value from 1 to 5 in the Simulink block diagram.
 - Change the constant block value (in the Simulink block diagram) from 0 to 1.

This starts the standalone simulation. Then, it starts Simulink's external mode and allows you to change model parameters in the generated code automatically. Although it is not obvious on your computer, the underlying mechanism allows parameter changes to generated code even if running on target hardware. The only requirement in such an application is that your

external hardware must provide TCP/IP support. For example, TCP/IP stack or library that you can link against during the build process.

Observe messages in the DOS window regarding updating of parameters.

When you complete this example, go to the **Simulink** menu or **External Mode Control Panel** and select **Disconnect**, and then select **Normal** to resume normal simulation mode.

Loop Rolling Through a TLC File

The following `%roll` code is taken from the `Outputs` function of `timestwo.tlc`.

```
%% Function: Outputs =====
%% Abstract:
%%      Y = 2 * U
%%
%function Outputs(block, system) Output
%assign fcnName = ParamSettings.FunctionName
/* %<Type> Block: %<Name> (%<fcnName>) */
%assign rollVars = ["U", "Y"]
%roll sigIdx=RollRegions, lcv=RollThreshold, block, ...
    "Roller", rollVars

%assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
%assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)

%<y> = 2.0 * %<u>;

%endroll

%endfunction %% Outputs
```

Arguments for %roll

The lines between `%roll` and `%endroll` may be either repeated or rolled. The key to understanding `%roll` is in the arguments. Look at the `%roll` line in detail.

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

The first argument is `sigIdx`. TLC uses `sigIdx` to specify the appropriate index into a vector that is used in the generated code. Suppose you have a scalar signal, TLC looks at the `.rtw` file and determines that only a single line of code is used and loop rolling is not needed. In this case, it will set `sigIdx` to 0 to be used to access only the first element of a vector.

Look at the second argument of `%roll`, `lcv`, which is generally set in the `%roll` line as `lcv = RollThreshold`. `RollThreshold` is a globally used threshold with the default value of 5. Therefore, where more than five contiguous androllable variables exist, TLC will collapse the lines nested between `%roll` and `%endroll` into a loop. If less than five contiguousrollable variables are found (for example, you build a block diagram where your block has only 4 inputs), `%roll` will *not* create a loop and instead will produce individual lines of code.

The third argument should just be `block`. This tells TLC that it is operating on block objects. Any TLC code for an S-function simply uses this argument as shown.

The fourth argument of `%roll` is a string `Roller`. Use this as is.

The fifth argument is `rollVars`. `rollVars` tells TLC which types of items should be rolled. Obvious choices include input signals, output signals, and parameters. It is not necessary to use all of them. In a previous line, `rollVars` is defined using `%assign`.

```
%assign rollVars = [ "U", "Y", "P" ]
```

This list tells TLC that it is rolling on input signals, output signals, and parameters.

Input Signals, Output Signals, and Parameters

Look at the line that appear between `%roll` and `%endroll`.

```
%assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
```

The first argument, 0, corresponds to the input port index for a given block. The first input port has index 0. The second input port has index 1, and so on.

The second argument is reserved for advanced use. For now, specify the second argument as an empty string. In advanced applications, you could define your own variable name to be used as an index with `%roll`. In such a case, TLC declares this variable as an integer in an appropriate location in the generated code.

The third argument to the function `LibBlockInputSignal` is `lcv`. As described previously, `lcv = RollThreshold` was set in `%roll` to indicate that we will loop if `RollThreshold` (default value of 5) is exceeded. For example, if there are 6 contiguous inputs into the block, they will be rolled.

The fourth argument, `sigIdx`, allows TLC to perform some magic. In the event that the `RollThreshold` is *not* exceeded (for example, if the block is only connected to a scalar input signal) TLC will not roll it into a loop. Instead, TLC will insert an integer value on the appropriate variable and place the corresponding line of code as “straight-line” code.

If the Roll Threshold is exceeded, (if there are 10 inputs to the block) TLC collapses the lines of code into a for loop and uses an index variable to index into lines within the for loop.

TLC Function Library Reference

Obsolete Functions	9-3
Target Language Compiler Functions	9-5
Input Signal Functions	9-10
Output Signal Functions	9-15
Parameter Functions	9-18
Block State and Work Vector Functions	9-22
Block Path and Error Reporting Functions	9-25
Code Configuration Functions	9-27
Sample Time Functions	9-30
Other Useful Functions	9-36
Advanced Functions	9-39

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*.

You can find examples using these functions in *matlabroot/toolbox/simulink/blocks/tlc_c* and *matlabroot/toolbox/simulink/blocks/tlc_ada*. The corresponding MEX S-function source code is located in *matlabroot/simulink/src*. M-file S-function 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
Li bBl ockOut port Locat i on	Li bBl ockDstSi gnal Locat i on
Li bCacheGl obal PrmData	Use the block function Bl ockI nst anceData.
Li bCont i nuousSt ate	Li bBl ockCont i nuousSt ate
Li bControl Port I nputSi gnal	Li bBl ockSrcSi gnal Locat i on
Li bData I nput Port Wi dth	Li bBl ockI nputSi gnal Wi dth
Li bData Out put Port Wi dth	Li bBl ockOut putSi gnal Wi dth
Li bDefi neI Work	Specifying IWork names is now supported via the mdl RTW function in your C-MEX S-function.
Li bDefi nePWork	Specifying PWork names is now supported via the mdl RTW function in your C-MEX S-function.
Li bDefi neRWork	Specifying RWork names is now supported via the mdl RTW function in your C-MEX S-function.
Li bDi screteSt ate	Li bBl ockDi screteSt ate
Li bExt ernal ResetSi gnal	Li bBl ockI nputSi gnal
Li bMapSi gnal Source	FcnMapDataTypedSi gnal Source
Li bMaxBl ockI OWi dth	Function is not used in the Real-Time Workshop.
Li bMaxData I nput Port Wi dth	Function is not used in the Real-Time Workshop.

Obsolete Function	Equivalent Replacement Function
LibMaxDataOutputPortWidth	Function is not used in the Real-Time Workshop.
LibPathName	LibGetBlockPath, LibGetFormattedBlockPath
LibPrevZCState	LibBlockPrevZCState
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, otherwise 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>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>Most functions that accept a <code>sigIdx</code> argument, accept it in an overloaded form where <code>sigIdx</code> can be:</p> <ul style="list-style-type: none">• 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.• An <i>id-num</i> usually of the form (see “Overloading sigIdx” on page 9-7):<ul style="list-style-type: none">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.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>The <code>idx</code> name is used when referring to a state or work vector.</p> <p>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 within a <code>%roll</code> directive. If they are used within a <code>%roll</code> directive, <code>ucv</code> is generally specified as <code>" "</code>, <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
<code>paramIdx</code>	Parameter index. Sometimes referred to as the parameter element index. The handling of this parameter is very similar to <code>sigIdx</code> (i.e., it can be #, <code>re#</code> , or <code>im#</code>).
<code>stateIdx</code>	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` some times written as `idx`) can be overloaded when passed to most library functions. Suppose we interested in element 3 of a signal, `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`.

<code>sigIdx</code>	Complex	Function Returns	Example	Data Type
<code>"re3"</code>	yes	Real part of element 3	<code>u0[2].re</code>	<code>real_T</code>
<code>"im3"</code>	yes	Imaginary part of element 3	<code>u0[2].im</code>	<code>real_T</code>
<code>"3"</code>	yes	Complex container of element 3	<code>u0[2]</code>	<code>creal_T</code>
<code>3</code>	yes	Complex container of element 3	<code>u0[2]</code>	<code>creal_T</code>
<code>"re3"</code>	no	Element 3	<code>u0[2]</code>	<code>real_T</code>
<code>"im3"</code>	no	<code>" "</code>	N/A	N/A

sigIdx	Complex	Function Returns	Example	Data Type
"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 i dx, whether the signal is complex, and what the function that uses i dx returns.

sigIdx	Complex	Function Returns
"re3"	yes	Real part of element i
"i m3"	yes	Imaginary part of element i
"3"	yes	Complex container of element i
3	yes	Complex container of element i
"re3"	no	Element i
"i m3"	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 si gI dx in the above examples and both lcv and si gI dx 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 value) for an expression. The block input signal may be coming 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 this function 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 a 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 (ucv, lcv, and sigIdx) Handling

Consider:

Function (case 1, 2, 3)	Example Return Value
<code>Li bBl ockI nputSi gnal (0, "i ", "", si gIdx)</code>	<code>rtB. bl ockname[i]</code>
<code>Li bBl ockI nputSi gnal (0, "", lcv, si gIdx)</code>	<code>u0[i 1]</code>
<code>Li bBl ockI nputSi gnal (0, "", lcv, si gIdx)</code>	<code>rtB. bl ockname[0]</code>

The value returned depends on what the input signal is connected to in the block diagram and how the function is being invoked (e.g., in a `%roll` or directly). In the above example, case 1 occurs when an explicit call is made with the `ucv` set to `"i "`. Cases 2 and 3 receive the same arguments, `lcv` and `sigIdx`, however they produce different return values. Case 2 occurs when `Li bBl ockI nputSi gnal` is called within a `%roll` directive and the current roll region is being rolled. Case 3 occurs when `Li bBl ockI nputSi gnal` is called within a `%roll` directive and the current roll region is not being rolled.

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_mulptiport.tlc`.

Example 1

To assign the outputs of a block to be the square of the inputs, you could use

```
%assign rollVars = ["U", "Y"]
%roll sigIdx = RollRegions, lcv = RollThreshold, block, "Roller", rollVars
%assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
%assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
%<y> = %<u> * %<u>;
%endroll
```

This would be appropriate for a block with a single input port and a single output port.

Example 2

Suppose we have a block with multiple input ports where each port has width greater than or equal to 1 and at least one port has width 1. Setting the output signal to the sum of the squares of all the input signals gives

```
%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`.

See function in `matlabroot/rtw/c/tlc/blkiolib.tlc` or `matlabroot/rtw/ada/tlc/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.

In Ada, this function returns the identifier and the caller must append the 'Address attribute.

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/blocklib.tlc* or *matlabroot/rtw/ada/tlc/blocklib.tlc*.

LibBlockInputSignalDataTyped(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/blocklib.tlc* or *matlabroot/rtw/ada/tlc/blocklib.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/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockInputSignalDimensions(portIdx)

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

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockInputSignalIsComplex(portIdx)

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

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockInputSignalIsFrameData(portIdx)

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

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockInputSignalNumDimensions(portIdx)

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

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockInputSignalWidth(portIdx)

Returns the width of the specified block input port index.

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.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 where the output signal destination, `LibBlockOutputSignal` returns the appropriate reference to a block output signal.

The returned value is a valid `lvalue` (left 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.

The Real-Time Workshop tracks when a variable (e.g., signals and parameters) 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/blocklib.tlc` or `matlabroot/rtw/ada/tlc/blocklib.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.

In Ada, this function returns the identifier and the caller must append the 'Address attribute.

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/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockOutputSignalDataTypeId(portIdx)

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

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

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockOutputSignalDataTypeName(portIdx, reim)

Returns the type name string (e.g., *int_T*, ... *creal_T*) of the data type corresponding to the specified block output port.

Specify the *reim* argument as "" if you want the complete signal type name. For example, if *reim*="" and the first output port is real and complex, the data type name placed in *dtname* will be *creal_T*.

```
%assign dtname = LibBlockOutputSignalDataTypeName(0x, "")
```

Specify the *reim* argument as *tReal Part* if you want the raw element type name. For example, if *reim*=*tReal Part* and the first output port is real and complex, the data type name returned will be *real_T*.

```
%assign dtname = LibBlockOutputSignalDataTypeName(0, tReal Part)
```

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockOutputSignalDimensions(portIdx)

Returns the dimensions of specified block output port.

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockOutputSignalIsComplex(portIdx)

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

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockOutputSignalIsFrameData(portIdx)

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

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockOutputSignalNumDimensions(portIdx)

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

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockOutputSignalWidth(portIdx)

Returns the width of specified block output port.

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.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 rlc v 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/paramlib.tlc* or *matlabroot/rtw/ada/tlc/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., rlc v and clcv should both be the empty string).

See function in *matlabroot/rtw/c/tlc/paramlib.tlc* or *matlabroot/rtw/ada/tlc/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-hand side expression value). 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	1	scalar	4.55	no
2	1	1	scalar	4.55	no
3	0	1	vector	4.55	no
4	1	1	vector	p_Gain[i]	yes

Case	Rolling	Inline Parameter	Type	Result	Required In Memory
5	0	0	scalar	rtP. blk. Gain	no
6	0	0	scalar	rtP. blk. Gain	no
7	0	0	vector	rtP. blk. prm[0]	no
8	0	0	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 M 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/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 entirely by using `LibBlockParameterAddr()`.

See function in `matlabroot/rtw/c/tlc/paramlib.tlc` or `matlabroot/rtw/ada/tlc/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/paramlib.tlc` or `matlabroot/rtw/ada/tlc/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/paramlib.tlc` or `matlabroot/rtw/ada/tlc/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/paramlib.tlc` or `matlabroot/rtw/ada/tlc/paramlib.tlc`.

LibBlockParameterIsComplex(param)

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

See function in `matlabroot/rtw/c/tlc/paramlib.tlc` or `matlabroot/rtw/ada/tlc/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/paramlib.tlc` or `matlabroot/rtw/ada/tlc/paramlib.tlc`.

Block State and Work Vector Functions

LibBlockContinuousState(ucv, lcv, idx)

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

See function in *matlabroot/rtw/c/tlc/blocklib.tlc* or *matlabroot/rtw/ada/tlc/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/blocklib.tlc* or *matlabroot/rtw/ada/tlc/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/blocklib.tlc* or *matlabroot/rtw/ada/tlc/blocklib.tlc*.

LibBlockDWorkDataTypeId(dwork)

Returns the data type ID of specified block DWORK.

See function in *matlabroot/rtw/c/tlc/blocklib.tlc* or *matlabroot/rtw/ada/tlc/blocklib.tlc*.

LibBlockDWorkDataTypeName(dwork, reim)

Returns the data type name of specified block DWORK.

See function in *matlabroot/rtw/c/tlc/blocklib.tlc* or *matlabroot/rtw/ada/tlc/blocklib.tlc*.

LibBlockDWorkIsComplex(dwork)

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

See function in *matlabroot/rtw/c/tlc/blocklib.tlc* or *matlabroot/rtw/ada/tlc/blocklib.tlc*.

LibBlockDWorkName(dwork)

Returns the name of the specified block DWORK.

See function in *matlabroot/rtw/c/tlc/blocklib.tlc* or *matlabroot/rtw/ada/tlc/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/blocklib.tlc* or *matlabroot/rtw/ada/tlc/blocklib.tlc*.

LibBlockDWorkWidth(dwork)

Returns the width of the specified block DWORK.

See function in *matlabroot/rtw/c/tlc/blocklib.tlc* or *matlabroot/rtw/ada/tlc/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/blocklib.tlc* or *matlabroot/rtw/ada/tlc/blocklib.tlc*.

LibBlockIWork(iwork, ucv, lcv, idx)

Returns a string corresponding to the specified block IWORK element.

See function in *matlabroot*/rtw/c/tlc/blocklib.tlc or *matlabroot*/rtw/ada/tlc/blocklib.tlc.

LibBlockMode(ucv, lcv, idx)

Returns a string corresponding to the specified block MODE element.

See function in *matlabroot*/rtw/c/tlc/blocklib.tlc or *matlabroot*/rtw/ada/tlc/blocklib.tlc.

LibBlockPWork(pwork, ucv, lcv, idx)

Returns a string corresponding to the specified block PWORK element.

See function in *matlabroot*/rtw/c/tlc/blocklib.tlc or *matlabroot*/rtw/ada/tlc/blocklib.tlc.

LibBlockRWork(rwork, ucv, lcv, idx)

Returns a string corresponding to the specified block RWORK element.

See function in *matlabroot*/rtw/c/tlc/blocklib.tlc or *matlabroot*/rtw/ada/tlc/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/utillib.tlc or *matlabroot*/rtw/ada/tlc/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 Appendix B, "TLC Error Handling."

See function in *matlabroot*/rtw/c/tlc/utillib.tlc or *matlabroot*/rtw/ada/tlc/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/utllib.tlc* or *matlabroot/rtw/ada/tlc/utllib.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 `hilit_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/utllib.tlc* or *matlabroot/rtw/ada/tlc/utllib.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 a `' //'` 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/utllib.tlc* or *matlabroot/rtw/ada/tlc/utllib.tlc*.

Code Configuration Functions

LibAddToModelSources(newFile)

For blocks, this function is generally called from `BlockTypeSetup`. This function adds a filename 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.

See function in `matlabroot/rtw/c/tlc/commonhdrlib.tlc` or `matlabroot/rtw/ada/tlc/commonhdrlib.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`.

C

This function caches `#define` statements for inclusion in `model.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`.

Example.

```
%openfile buffer
#define INTERP(x, x1, x2, y1, y2) ( y1+((y2-y1)/(x2-x1))*(x-x1) )
#define this that
%closefile buffer
%<LibCacheDefine(buffer)>
```

Ada

This function caches definitions for inclusion in `model.adb`. The Ada utility functions and procedures are placed in the generated `model.adb` package body immediately preceding `Mdl_Start`.

See function in `matlabroot/rtw/c/tlc/cachelib.tlc` or `matlabroot/rtw/ada/tlc/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* or *model.ads*.

C Example

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

See function in *matlabroot/rtw/c/tlc/cachelib.tlc* or *matlabroot/rtw/ada/tlc/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/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)>
```


For Ada, the equivalent statements are placed in the Ada package specification.

See function in *matlabroot/rtw/c/tlc/cachelib.tlc* or *matlabroot/rtw/ada/tlc/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 the Ada package specification if the language is Ada.

Example

```
%openfile buffer
typedef foo bar;
%closefile buffer
%<LibCacheTypedefs(buffer)>
```

See function in *matlabroot/rtw/c/tlc/cachelib.tlc* or *matlabroot/rtw/ada/tlc/cachelib.tlc*.

Sample Time Functions

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 global TID = LibGetGlobalTIDFromLocalSFcnTID(2)
```

or

```
%assign global TID =  
LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx4")
```

```
%assign period =  
CompiledModel.SampleTime[global TID].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/utillib.tlc or *matlabroot*/rtw/ada/tlc/utillib.tlc.

LibGetNumSFcnSampleTimes(block)

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

See function in *matlabroot*/rtw/c/tlc/utillib.tlc or *matlabroot*/rtw/ada/tlc/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 LibISFcnSampleHt.

Note This is useful primarily in the context of S-functions that specify an inherited sample time.

See function in *matlabroot/rtw/c/tlc/utllib.tlc* or *matlabroot/rtw/ada/tlc/utllib.tlc*.

LibGetTaskTimeFromTID(block)

Returns the string "ssGetT(S)" if the block is constant or the system is single rate and "ssGetTaskTime(S, tid)" otherwise. In both cases, S is the name of the SimStruct.

See function in *matlabroot/rtw/c/tlc/utllib.tlc* or *matlabroot/rtw/ada/tlc/utllib.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/utllib.tlc* or *matlabroot/rtw/ada/tlc/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/utillib.tlc* or *matlabroot/rtw/ada/tlc/utillib.tlc*.

LibsSFcnSampleHit(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 `LibsSFcnSampleHit(2)` will return the code to check for a sample hit on the 3rd S-function block sample time.
- Consider a multirate S-function block with three input and eight output sample times. The call `LibsSFcnSampleHit("InputPortIdx0")` returns the code to check for a sample hit on the first input port. The call `LibsSFcnSampleHit("OutputPortIdx7")` returns the code to check for a sample hit on the eight output port.

See function in *matlabroot/rtw/c/tlc/utillib.tlc* or *matlabroot/rtw/ada/tlc/utillib.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/utllib.tlc or *matlabroot*/rtw/ada/tlc/utllib.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 multi-rate 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/utllib.tlc or *matlabroot*/rtw/ada/tlc/utllib.tlc.

Other Useful Functions

LibCallFCSS(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 generates 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 LibIsEqual(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/syslib.tlc* or *matlabroot/rtw/ada/tlc/syslib.tlc*.

LibGetDataTypeComplexNameFromId(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/dtypelib.tlc* or *matlabroot/rtw/ada/tlc/dtypelib.tlc*.

LibGetDataTypeEnumFromId(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/dtypelib.tlc* or *matlabroot/rtw/ada/tlc/dtypelib.tlc*.

LibGetDataTypeNameFromId(id)

Returns the data type name corresponding to a data type ID.

See function in *matlabroot/rtw/c/tlc/dtypelib.tlc* or *matlabroot/rtw/ada/tlc/dtypelib.tlc*.

LibGetT0

Returns a string to access the absolute time. You should only use this function to access time.

Calling this function causes the global flag `CompiledModel.NeedAbsoluteTime` to be set to 1. If this flag isn't set and you manually accessed time, the generated code will not compile.

This function is the TLC version of the `SimStruct` macro, `ssGetT`.

See function in *matlabroot/rtw/c/tlc/utllib.tlc* or *matlabroot/rtw/ada/tlc/utllib.tlc*.

LibIsComplex(arg)

Returns 1 if the argument passed in is complex, 0 otherwise.

See function in *matlabroot/rtw/c/tlc/utllib.tlc* or *matlabroot/rtw/ada/tlc/utllib.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 the 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/syslib.tlc* or *matlabroot/rtw/ada/tlc/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/dtypelib.tlc* or *matlabroot/rtw/ada/tlc/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/dtypelib.tlc* or *matlabroot/rtw/ada/tlc/dtypelib.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 implementation 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>;  
  
%i f (Li bBl ockI nputSi gnal BufferDstPort (0) != - 1)  
    }  
%endi f
```

Note that this example could have equivalently used `(Li bBl ockI nputSi gnal BufferDstPort (0) == 0)` as the boolean condition for the `%i f` statements since there is only one output port.

See function in *matlabroot*/rtw/c/tlc/blkiolib.tlc or *matlabroot*/rtw/ada/tlc/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/blkiolib.tlc or *matlabroot*/rtw/ada/tlc/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/blkiolib.tlc or *matlabroot*/rtw/ada/tlc/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/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/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/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockOutputSignalsValidLValue(portIdx)

Returns 1 if the specified block output port signal can be used as a valid left hand 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/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockOutputSignalStorageClass(portIdx)

Returns the storage class of the block's specified output signal. The storage class can be "Auto", "ExportedSignal", "ImportedExtern", or "ImportedExternPointer".

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

LibBlockOutputSignalStorageTypeQualifier(portIdx)

Returns the storage type qualifier of the block's specified output signal. The type qualifier can be anything entered by the user such as "const". The default type qualifier is "Auto", which means do the default action.

See function in *matlabroot/rtw/c/tlc/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.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.
"External Input"	If external input (root input).
"Ground"	If unconnected or connected to ground.
"FcnCall"	If function-call output.
0	If not unique (i.e., sources to a Merge block or is a reused signal due to block I/O optimization).

Example

If you want to find the block that drives the second input on the first port of the current block, then, assign the input signal of this source block to the variable *y*. The following code fragment does exactly this.

```
%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/blkolib.tlc* or *matlabroot/rtw/ada/tlc/blkolib.tlc*.

LibBlockSrcSignalIsDiscrete(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/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/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/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.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/blkiolib.tlc* or *matlabroot/rtw/ada/tlc/blkiolib.tlc*.

model.rtw

model.rtw File Contents	A-2
model.rtw Changes Between Real-Time Workshop 3.0 and 4.0	A-6
General Information and Solver Specification	A-11
RTWGenSettings Record	A-13
Data Logging Information	A-14
Data Structure Sizes	A-16
Sample Time Information	A-18
Data Type Information	A-20
Block Type Counts	A-21
Model Hierarchy	A-22
External Inputs and Outputs	A-25
Data Store Information	A-27
Block I/O Information	A-28
Data Type Work (DWork) Information	A-33
State Mapping Information	A-35
Block Record Defaults	A-36
Parameter Record Defaults	A-37
Data and Control Port Defaults	A-38
Model Parameters Record	A-40
System Record	A-43
Stateflow Record	A-56
Model Checksums	A-57
Block Specific Records	A-58
Linear Block Specific Records	A-77

model.rtw File Contents

This appendix describes the contents of the *model.rtw* file, which is created from your block diagram during the Real-Time Workshop build procedure, and processed by the Target Language Compiler. The contents of the *model.rtw* file is a *compiled* version of your block diagram. The *model.rtw* file contains all the information necessary to define behavioral properties of the model for the purpose of generating code. Most graphical model information is excluded from the *model.rtw* file.

This appendix is provided so that you can modify the existing code generation or even create a new *code generator* to suit your needs. The general format of the *model.rtw* file is

```
CompiledModel {  
    <TLC variables and records describing the compiled model>  
}
```

Understanding the model.rtw File

You need to understand the basic format of the *model.rtw* file if you are writing a TLC file for an S-function (i.e., inlining the S-function). You do not, however, need to know all the details about the *model.rtw* file. For the purpose of inlining an S-function, you only need to understand the concepts of the *model.rtw* file and how to access the information using the Target Language Compiler.

Items such as signal connectivity and obtaining input and output connections for your S-function are contained within the *model.rtw* file using mapping tables. Processing this information directly in the Target Language Compiler is difficult and will not remain compatible between releases of our tools. To simplify writing TLC files for S-functions and provide compatibility between, many library functions (which start with the prefix *LibBlockInputSignal*) are provided. For example to access your inputs to your S-function, you should use *LibBlockInputSignal*.

When the Target Language Compiler calls the various functions that exist in your TLC file, the *Block* record for your S-function will be scoped. In this case, you have access to the *Parameters* and *ParamSettings* records shown in the *Block Type: S-Function* section.

If your S-function has an mdl RTW method, then you can control several fields within the Block record. For example, you can use the function `ssWriteRTWParamSettings` to have `rtwgen` create a `SFcnParameterSettings` record containing the “nontunable” (see `ssSetSFcnParamTunable` in the Simulink book *Writing S-Functions*) parameter values in the Block record for your S-function. There are several other functions available to mdl RTW for adding information to the `model.rtw` file. See `matlabroot/simulink/src/sfuntmpl.doc` for more information.

In addition, there are many Target Language Compiler library functions available to help you inline S-functions. See Chapter 9, “TLC Function Library Reference,” for a complete list of Target Language Compiler library functions.

Note The contents of the `model.rtw` file may change from release to release. The MathWorks will make every effort to keep the `model.rtw` file compatible with previous releases. We cannot, however, guarantee that the file will be compatible between major enhancement releases. We will always try to maintain compatibility for the MathWorks-provided Target Language Compiler library functions (Lib*). We will document any improvements/changes to the library functions.

How TLC Operates on a Record File

To understand the format of the `model.rtw` file, you need to understand how the Target Language Compiler operates on a record (database) file, e.g., `model.rtw`. The `model.rtw` contains parameter value pairs, records, lists, default records, and parameter records.

An example of a parameter value pair (or field) is

```
Signal "velocity"
```

which specifies that the field (or variable) `Signal` contains the value “velocity”. You can place this field in a record named `Signal` with

```
Signal {  
    Signal "velocity"  
}
```

Accessing Record Fields. To access fields within a record, use the dot operator. For example, `Signal.SignalLabel` accesses the signal label field of the `Signal` record.

Changing Scope. You can change the local scope to any record in the Target Language Compiler using the `with` directive. This allows for both relative and absolute scoping. The Target Language Compiler first checks for the item being accessed in the local scope; if the item is not there, it then searches the global name pool (global scope).

Creating a List. The Target Language Compiler creates a list by contacting several records. For example,

```
NumSignal s 2
Signal {
    SignalLabel "velocity"
}
Signal {
    SignalLabel "position"
}
```

This code creates a parameter called `NumSignal s` that specifies the length of the list. This is useful when using the `foreach` directive. To access the second signal, use `Signal [1]`. Note, the first index in a Target Language Compiler list is 0.

You can create a default record by appending the word `Defaults` to the record name. For example,

```
SignalDefaults {
    ComplexSignal no
}
Signal {
    SignalLabel "velocity"
}
```

An access to the field `Signal.ComplexSignal` returns `no`. The Target Language Compiler first checks the `Signal` record for the field (parameter) `ComplexSignal`. Since it does not exist in this example, the Target Language Compiler searches for the field `SignalDefaults.ComplexSignal`, which has the value `no`. (If `SignalDefaults.ComplexSignal` did not exist, it would generate an error.)

A parameter record is a record named `Parameter` that contains, at a minimum, the fields `Name` and `Value`. The Target Language Compiler automatically promotes the parameter up one level and creates a new field containing `Name` and `Value`.

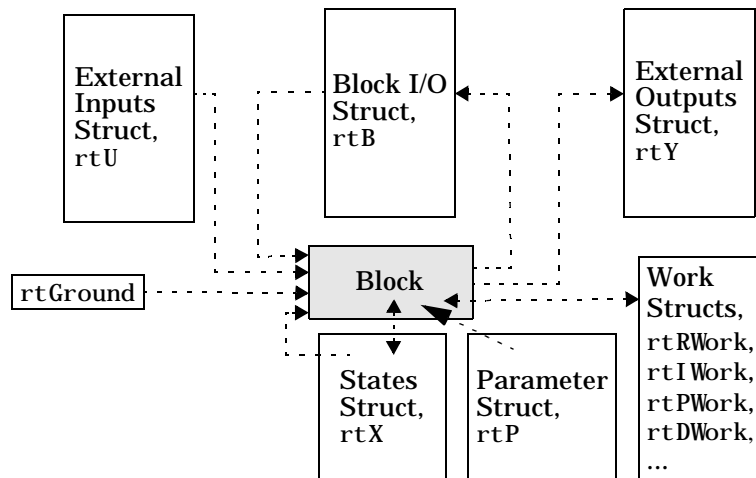
For example,

```
Block {
  Parameter {
    Name Velocity
    Value 10.0
  }
}
```

You can access the `Velocity` parameter using `Block.Velocity`. The value returned is 10.0.

General model.rtw Concepts

The layout of the `model.rtw` file is based around the structure of Simulink models. Conceptually, the model consists of systems and blocks — blocks read their input, manage their states, and write their output. This figure outlines the basic object-oriented view of a block. At the model level, there are well-defined working areas such as the block I/O (`rtB`) vector.



model.rtw Changes Between Real-Time Workshop 3.0 and 4.0

Several changes have been made to the `model.rtw` file. These changes:

- Provide more functionality
- Reduce generated code size
- Increase generated code efficiency
- Faster code generation

model.rtw Differences

- The `StatesMap` was updated to reduce the size of the `model.rtw` file, making code generation faster. The `StatesMap` now contains rows for states that go to a port (direct port connections). For example, `Signal Src X4` means the fifth row in the `StatesMap` only, not the fifth index.4. Note, the `StatesMap` now only contains the a mapping for the double, non-complex continuous and discrete states.
- S-function changes:
 - Added `UsingUPtrs ParamSetting` for Level 2 C-MEX S-functions. This is needed to support the `ssSetInputPortRequiredContinuous` flag for non-inlined Level 2 C-MEX S-functions.
 - If the number of input ports for a Level 1 S-function is zero, then the `ParamSetting.Continuous` field now writes out as "" instead of "yes".
- Removed `Slopes` parameter from 1D Look-Up table. Slopes for the 1D look-up are computed at runtime.
- Removed `DiscStatesDataTypeIdx` and `DiscStatesComplexSignal`. These are now handled by the data type work vector fields.
- Added support for frames. A frame signal is a multi-element signal consisting of multiple samples for a given time point. Frame signals are typically used in signal processing algorithms. Specifically:
 - A `FrameData` field was added to each block input port record.
 - A `DataOutputPortFrameData` field was added to the block record. This was not added to block output port record because this record is specific to a buffer and not to a block. In the case of frame data, the same buffer could have different frame interpretations based on the block they feed into.

- Added to the BlockOutputDefaults record:

NumReusedBlockOutputs 0

NumMergedBlockOutputs 0

- Combined discrete states with the data type work vector handling. Updated DataLoggingOpts to reflect discrete states are now a part of DWork. Specifically:

- Removed

CompiledModel.NumDiscStates

CompiledModel.Modes

CompiledModel.NumRWork

CompiledModel.NumIWork

CompiledModel.NumPWork

CompiledModel.NumDWork

CompiledModel.NumDWorkRecords

- Renamed the DWork record as follows.

old	new
<pre> DWorkRecords { DWorkRecordDefaults { DataTypeIdx ComplexSignal UsedAsDState } NumDWorkRecords DWorkRecord { BlockIdentifier Name Width DataTypeIdx ComplexSignal UsedAsDState DWorkSrc } : : } </pre>	<pre> DWorks { DWorkDefaults { DataTypeIdx ComplexSignal UsedAs } NumDWorks DWork { Identifier Width DataTypeIdx ComplexSignal UsedAs SigSrc } : : } </pre>

- Changed the following fields from the block record.

	used to be	changed to
Block. DiscStates	[width, vectIdx]	[width, dWorkIdx]
Block. RWork	[width, vectIdx]	[width, dWorkIdx]
Block. IWork	[width, vectIdx]	[width, dWorkIdx]
Block. PWork	[width, vectIdx]	[width, dWorkIdx]
Block. ModeVector	[width, vectIdx]	[width, dWorkIdx]

where vectIdx was the vector index into the corresponding vector, while recIdx is now the index of the corresponding DWork in the model-wide DWorks record.

- Changed "Block. DWork[xx]. RecordIdx" to "Block. Dwork[xx]. RecordIndex".
- To support the addition of run-time parameters, the block Parameter records are now the last items in the Block records (i.e., parameter settings and so on come before block Parameter records and the parameter count vector.)
- To support parameter pooling, moved the writing of the Model Parameters record above the System records. Several new fields were added to the Model Parameters record.
- Added support for block parameter aliases. For example, lock records now look like

```

NumParameters 2
Parameter {
}
Parameter {
}
P1Name Parameter[0]
P2Name Parameter[1]

```

- Added support for block parameter value aliases when there is a one-to-one mapping with the model parameters record. Parameter records now have

```

Parameter {
    Value CompiledModel.ModelParameters.Parameter[#].Value
}

```

when a mapping can be located between the values in Parameter records and the Model Parameters record. This was done to speed up code

generation and reduce the size of the *model.rtw* file when we can locate a mapping between the values in Parameter records and the Model Parameters record.

- Added support for matrix signals. All parameters are assumed to be in column-major ordering and several blocks were updated to reflect this. Matrix support involved adding a Dimensions field to many records. Specifically, we now:
 - Write actual block and model parameter dimensions, i.e., updated the Parameter records in the Block and Model Parameters Parameter records.
 - Write matrix dimensions for data store memory blocks, i.e., updated the DataStores record.
 - Write block output port dimensions only if the output signal is a matrix, i.e., updated RootSignals or Subsystem record.
 - Write external input dimensions for matrix signals, i.e., updated ExternalInputs record.
 - Write external output dimensions for matrix signals, i.e., updated ExternalOutputs record.
 - Write block output dimensions for matrix signals, i.e., updated BlockOutput record.
 - Write data input port dimensions for matrix signals, i.e., updated DataInputPorts record.
 - Write data output port dimensions for matrix signal, i.e., updated DataOutputPort record.

Matrix support also required eliminating the notion of number of data logging rows. This required the renaming of the following blocks.

- Scope:
 - Renamed "MaxRows" to "MaxDataPoints"
 - Renamed "LimitMaxRows" to "LimitDataPoints"
 - Renamed "Matrix" option in SaveFormat parameter to "Array" format
- ToWorkspace:
 - Renamed "Buffer" to "MaxDataPoints"
 - Renamed "Matrix" option in SaveFormat parameter to "Array" format

- FromWorkspace:
 - Renamed "Matrix" option in Data format parameter to "Array" format
- Added DataOutputPort records to the Block records.
- Added ObjectProperties for signals specified in the Unified Data Repository to the BlockOutputs record.
- Many of the block records changed to support matrices.
- Some blocks records were updated to add data type support to them.

General Information and Solver Specification

When generated, each *model.rtw* contains some general information including the model name, the date when the *model.rtw* file was generated, the version number of the Real-Time Workshop that generated the *model.rtw* file, and so on. In addition, the Target Language Compiler takes information specific to the solver and solver parameters from the **Simulink Parameters** dialog box and places it into the *model.rtw* file.

The following describes the first part of the *model.rtw* file - general information (e.g., model name) and the solver specification.

Table A-1: Model.rtw General Information and Solver Specification

Variable/Record Name	Description
Name	Name of the Simulink model from which this <i>model.rtw</i> file was generated.
Version	Version of the <i>model.rtw</i> file.
GeneratedOn	Date and time when the <i>model.rtw</i> file was generated.
Solver	Name of solver as entered in the Simulink Parameters dialog box.
SolverType	FixedStep or VariableStep.
StartTime	Simulation start time as entered in the Simulink Parameters dialog box.
StopTime	Simulation stop time.
FixedStepOpts {	Only written if SolverType is FixedStep.
SolverMode	Either SingleTasking or MultiTasking.
FixedStep	Fundamental step size (in seconds) to be used.
TID01EQ	Either 0 or 1 indicating if the first two sample times are equal (1 if they are equal). This occurs where there is a continuous sample time and one or more discrete sample times and the fixed-step size is equal to the fastest discrete sample time.
}	
VariableStepOpts {	Only written if SolverType is VariableStep. These variable step options are used by the Simulink Accelerator and S-function targets.
RelTol	Relative tolerance.
AbsTol	Absolute tolerance.

Table A-1: Model.rtw General Information and Solver Specification (Continued)

Variable/Record Name	Description
Refine	Refine factor.
MaxStep	Maximum step size.
Initial Step	Initial step size.
MaxOrder	Maximum order for ode15s.
}	

RTWGenSettings Record

The `RTWGenSettings` record contains name/value pairs that are assigned via the system target file (e.g., `grt.tlc`). During an RTW build, the M-code in the system target file

```
rtwgensettings. fieldname1 = 'value1';  
...  
rtwgensettings. fieldnameN = 'valueN';
```

is executed and the `RTWGenSettings` that are created are set on the model using `set_param(model, 'RTWGenSettings', rtwgensettings)` by `make_rtw`. Note that the `rtwgensettings` field name values must be strings. Simulink can then use the `RTWGenSettings` to affect the compiled characteristics of the model and thus the contents of the `model.rtw`. The `model.rtw` will also contain the `RTWGenSettings` so that rest of the build process including TLC code can have access to them. Additional `RTWGenSettings` can be added and although Simulink will not use them, they will be written to the `model.rtw` file and can be used as desired by TLC code.

Table A-2: `Model.rtw` `RTWGenSettings` Record

Variable/Record Name	Description
<code>RTWGenSettings {</code>	
<code>BuildDirSuffix</code>	The string to append to the model name to create the build directory name.
<code>UsingMalloc</code>	Set to “yes” if the generated code will be using dynamic memory allocation.
<code>IsRSim</code>	Set to “yes” if this build is for the RSIM target.
<code>IsRTWSfcn</code>	Set to “yes” if this build is for the RTW S-function target.
<code>...</code>	Any additional string fields.
<code>}</code>	

Data Logging Information

The **Workspace I/O** page of the **Simulation Parameters** dialog box gives you the option of selecting whether to log data after executing model code generated from Real-Time Workshop. If you choose to log data, you must select one or more check boxes for Time, States, Output, or Final state. You can use the default variable names shown in the dialog box or replace these with your own name selections.

To conserve memory and/or file space when data logging, you can limit data collection to the final N-points of your simulation. Select the **Limit rows to last** check box and use the default value 1000. Using the default setting saves the selected workspace variables to a buffer of length 1000. If desired, you can provide another value for the total number of points per variable to store. You can also select to store the variables in one of three formats:

- Structure with time
- Structure
- Matrix

All data logging information corresponding to the **Workspace I/O** page is placed within the `CompiledModel.DataLoggingOpts` record. This record may change with future enhancements to the **Workspace I/O** page. It is intended to be used in conjunction with the MathWorks-provided MAT-file logging utility file, `matlabroot/rtw/c/src/rtwlog.c`.

This table lists and describes the data logging information for `model.rtw`.

Table A-3: Model.rtw Data Logging Information

Variable/Record Name	Description
<code>DataLoggingOpts {</code>	Data logging record describing the settings of Simulink simulation (Parameters, workspace, I/O settings).
<code>SaveFormat</code>	"Matrix", "Structure", or "StructureWithTime".
<code>MaxRows</code>	Maximum number of rows or 0 for no limit.
<code>Decimation</code>	Data logging interval.
<code>TimeSaveName</code>	Name of time variable or "" if not being logged.
<code>StateSaveName</code>	Name of state variable or "" if not being logged.

Table A-3: Model.rtw Data Logging Information (Continued)

Variable/Record Name	Description
OutputSaveName	Name of output variable or "" if not being logged.
NumOutputSaveNames	Number of names in the OutputSaveName list.
FinalStateName	Name of final state variable or "" if not being logged.
StateSigSrc	<p>Only written if SaveFormat is not Matrix and either the states or the final states are being logged. This is an N-by-3 matrix with rows:</p> <p style="padding-left: 40px;">[sysIdx, blkIdx, blkStateIdx]</p> <p>giving the location of the signal to be logged as a state. sysIdx and blkIdx give the source (i.e., a Block record), which specifies the states that are logged. The blkStateIdx is used to identify which part of the block the state is coming from.</p> <p>The blkStateIdx will be:</p> <ul style="list-style-type: none"> • -2 if the logged signal is the continuous state vector • -1 if the logged signal is the discrete state vector • ≥ 0 implies the logged signal is the blkStateIdxDWork (data type work vector) of blkIdx block in sysIdx system <p>Note that sysIdx and blkIdx are valid even if blkStateIdx < 0, and N is the number of signals being logged as states.</p>
	}

Data Structure Sizes

The *model.rtw* file contains several fields that summarize the size of data structures required by a particular model. This information varies from model to model, depending on how many integrators are used, how many inputs and outputs, and whether states are continuous time or discrete time. In the case where continuous states exist within a model, you must use a solver, for example, ode5.

The *model.rtw* file provides additional information about the total number of work vector elements used for a particular model including RWork, IWork, PWork, and DWork (real, integer, pointer, and data type work vectors). The DWork vector field is a new addition that provides information for data type work vectors (to support types other than real_T). The *model.rtw* file also provides fields that contain summary information for all block signals, block parameters, and number of algebraic loops found throughout the model.

This table describes the data structure written to *model.rtw*.

Table A-4: Model.rtw Model Data Structure Sizes

Variable/Record Name	Description
NumModel Inputs	Sum of all root-level import block widths. This is the length of the external input vector, U.
NumModel Outputs	Sum of all root-level output block widths. This is the length of the external output vector, Y.
NumNonVirtualBlocksInModel	Total number of nonvirtual blocks in the model.
DirectFeedthrough	Does model require its inputs in the MdlOutput function (yes/no)?
NumContStates	Total number of continuous states in the model. Continuous states appear in your model when you use continuous components (i.e., an Integrator block) that have state(s) that must be integrated by a solver such as ode45.
NumModes	Length of the model mode vector (modeVect). The mode vector is used by blocks that need to keep track of how they are operating. For example, the discrete integrator configured with a reset port uses the mode vector to determine how to operate when an external reset occurs.
ZCFindingDisabled	Is zero-crossing event location (finding) disabled (yes/no)? This is always yes for fixed-step solvers.

Table A-4: Model.rtw Model Data Structure Sizes (Continued)

Variable/Record Name	Description
NumNonsampl edZCs	Length of the model nonsampled zero-crossing vectors. There are two vectors of this length: the zero-crossing signals (nonsampl edZCs), and the zero-crossing directions (nonsampl edZCdi rs). Nonsampled zero-crossings are derived from continuous signals that have a discontinuity in their first derivative. Nonsampled zero-crossings only exist for variable step solvers. The Abs block is an example of a block that has an intrinsic, nonsampled zero-crossing to detect when its input crosses zero.
NumZCEvents	Length of the model zero-crossing event vector (zcEvents).
NumDWork	Total number of data type work vector elements. This is the sum of the widths of all data type work vectors in the model.
NumDataStoreEl ements	Total number of data store elements. This is the sum of the widths of all data store memory blocks in your model.
NumBl ockSi gnal s	Sum of the widths of all output ports of all nonvirtual blocks in the model. This is the length of the block I/O vector, bl ockI0.
NumBl ockParams	Number of modifiable parameter elements (params). For example, the Gain block parameter contains modifiable parameter elements.
NumAl gebrai cLoops	Number of algebraic loops in the model.

Sample Time Information

The sample time information written to *model.rtw* file describes the rates at which the model executes. The Fundamental StepSize corresponds to the base rate for the fastest task in a model.

The InvariantConstants field is set as a result of the **Inline parameters** check box in the **Real-Time Workshop** page of the **Simulation Parameters** dialog box. This allows you to globally select whether or not parameter inlining is to be used in generated code. An inlined parameter results in a parameter value being hard-coded in the generated code. Consequently, this value cannot be altered by any parameter tuning method. You can override invariant constants on one or more selected signals by selecting **Tunable parameters** and specifying the variable name.

The SampleTime list contains all periodic rates found within your model. This list excludes constant and triggered sample times.

Table A-5: Model.rtw Sample Times

Variable/Record Name	Description
AllSampleTimesInherited	yes if all blocks in the model have inherited sample times, no otherwise.
InvariantConstants	yes if invariant constants (i.e., Inline parameters check box) is on, no if invariant constants is off.
FundamentalStepSize	Fundamental step size or 0.0 if one cannot be determined. Fixed step solvers will always have a nonzero step size. Variable step solvers may have a fundamental step size of 0.0 if one can not be computed from the sample times in the model.
SingleRate	yes if the model is single rate, no otherwise. A model is considered single-rate if either: <ul style="list-style-type: none">• The number of system sample times is one or triggered or constant.• The number of system sample times is two, the tids are 0 and 1, TID01EQ is true, and the system has no continuous states.

Table A-5: Model.rtw Sample Times (Continued)

Variable/Record Name	Description
NumSampleTimes	Number of sample times in the model followed by SampleTime info records, giving the TID (task ID), an index into the sample time table, and the period and offset for the sample time.
SampleTime {	One record for each sample time.
TID	Task ID for this sample time.
PeriodAndOffset	Period and offset for this sample time.
}	

Data Type Information

The DataTypes record provides a complete list of all possible data types that Simulink supports and the current mapping between the data types and a data type index. We strongly advise against adding to this list since future versions of Real-Time Workshop will extend this list and can result in new mappings of data types.

All data typing information is written in the following list of records within the DataTypes record. Individual records often specify an index into this table.

Table A-6: Model.rtw Data Types

Variable/Record Name	Description
DataTypes {	Data types defining all built-in (double, single, int8, uint8, int16, uint16, int32, uint32, bool, fcncll) and any blockset specific data types found within your model.
NumDataTypes	Integer, total number DataType records that follow. This includes one record for each built-in data type plus specific records for blocksets.
NumSLBuiltInDataTypes	Integer, number of Simulink built-in data types (less than or equal to NumDataTypes).
StrictBooleanCheckEnabled	Integer (0/1) Flag that indicating whether model had boolean data types enabled or not.
DataType {	One record for each data type in use.
SLName	ASCII data type name. Note, this SLName is not to be confused with the unmodified Simulink name parameters used elsewhere.
Id	Actual data type identifier which is used in Simulink. This is an integer that corresponds to the data type name.
}	
}	

Block Type Counts

The `model.rtw` contains *block type counts* that describe what blocks are in your model. This information is model dependent; it provides a summary of how many different types of blocks are used within a particular model as well as a list of records that summarize how many blocks of each block type are found within the particular model. Similarly, the total number of unique S-function names found within a model are reported as well as the count of occurrences for each S-function name. Information describing what types of blocks are used and how many of them there are is provided in the `BlockTypeCount` records.

This table lists all the available block type counts.

Table A-7: `Model.rtw` Block Type Counts

Variable/Record Name	Description
<code>NumBlockTypeCounts</code>	Number of different types of blocks in your model. A block type correlates to the MATLAB command <code>get_param('block', 'BlockType')</code> .
<code>BlockTypeCount {</code>	One record for each block type count.
Type	Type of the block (e.g., Gain).
Count	Total number of the given type.
<code>}</code>	
<code>NumSFunctionNameCounts</code>	Number of different S-functions used in your model. There will be one S-function for each MEX or M-function name specified in the S-function dialog. This will be less than or equal to the number of S-Function blocks in your model.
<code>SFunctionNameCount {</code>	One record for each S-function used in your model.
Name	S-function name.
Count	Total number of S-Function blocks using this S-function name.

Model Hierarchy

The *root* is the top level of the block diagram. The `RootSignal` record contains information about the makeup of signals within the root level. This includes indices to subsystems that are visible at the root level as well as the number of input and output signals that appear at the root level. The `Subsystem` records contain information about the number of virtual and nonvirtual subsystems contained in the model and identifiers for these.

Each subsystem in the model includes a system identifier, a name, and a set of indices to any additional child subsystems. Counts are also provided for the number of outputs for each subsystem and number, signal information, and total number of nonvirtual blocks within each subsystem.

The `RootSignal` and `Subsystem` records enable you to reconstruct the graphical model hierarchy. This is useful for third party monitoring and parameter tuning tools.

Table A-8: Model.rtw Model Hierarchy (Blocks, Signals, and Subsystems)

Variable/Record Name	Description
<code>RootSignal</code> {	Signal and block information in the root window.
<code>ChildSubsystemIndices</code>	Vector of integers specifying the subsystems that are directly contained within the root system. The indices index into the <code>CompiledModel.Subsystem</code> record list.
<code>NumSignals</code>	Number of block output signals (including virtual) blocks.
<code>Signal</code> {	One record for each block output signal (i.e., length of this list is <code>NumSignals</code>).
<code>Block</code>	[<code>sysIdx</code> , <code>blockIdx</code>] or block name string if a virtual block.
<code>SignalLabel</code>	Signal label if present.
<code>OutputPort</code>	[<code>outputPortIndex</code> , <code>outputPortWidth</code>].
<code>Dimensions</code>	Vector of the form [<code>nRows</code> , <code>nCols</code>] for the signal. Only written if number of dimensions is greater than 1.
<code>DataTypeIdx</code>	Index into the <code>CompiledModel.DataTypes</code> . <code>DataType</code> record list. Only written for nonvirtual blocks and if data type index is not 0 (the default data type of 0 corresponds to <code>real_T</code>).
<code>ComplexSignal</code>	yes/no: is the signal complex? Only written for nonvirtual blocks and if signal is complex.

Table A-8: Model.rtw Model Hierarchy (Blocks, Signals, and Subsystems) (Continued)

Signal Src	Vector of length outputPortWidth giving the location of the signal source.
}	
NumBlocks	Number of nonvirtual blocks in the root window of your model.
BlockSysIdx	System index for blocks in this subsystem.
BlockMap	Vector of length NumBlocks giving the blockIdx for each nonvirtual block in the root system.
}	
NumVirtualSubsystems	Total number of virtual (non-empty) subsystems in the model.
NumNonvirtualSubsystems	Total number of nonvirtual subsystems in the model.
Subsystem {	One record for each subsystem.
SysId	System identifier. Each subsystem in the model is given a unique identifier of the form S# (e.g., S3).
Name	Block name preceded with a <root> or <S#> token. The ID/Name values define an associative pair giving a complete mapping to the blocks full path name (e.g., <s2/gain1>).
SLName	Unmodified Simulink name. This is only written if it is <i>not</i> equal to Name. This will occur when generating code using the rtwgenStringMappings argument. For the Real-Time Workshop C targets, any block name that contains a new-line, '/*', or '*/' will have these characters remapped. For example, suppose the Simulink block name is <pre>my block name /* comment */</pre> The <i>model.rtw</i> file will contain <pre>Name "<Root>/my block name //+ comment +//" SLName "<Root>/my block name\n/* comment */"</pre>
Virtual	yes/no: Whether or not the subsystem is virtual.
ChildSubsystemIndices	Vector of integers specifying the subsystems that are directly contained within this subsystem. The indices index into the CompiledModel.Subsystem record list.
NumSignals	Number of block output signals (including virtual) blocks.
Signal {	One record for each block output signal (i.e., length of this list is NumSignals).
Block	[sysIdx, blockIdx] or block name string if a virtual block.

Table A-8: Model.rtw Model Hierarchy (Blocks, Signals, and Subsystems) (Continued)

OutputName	This field is written only if the signal is emanating from a subsystem. It is the Output block name corresponding to the output signal of a subsystem block.
SignalLabel	Signal label if present.
OutputPort	[outputPortIndex, outputPortWidth].
Dimensions	Vector of the form [nRows, nCols] for the signal. Only written if number of dimensions is greater than 1.
DataTypeIdx	Index into the CompiledModel.DataTypes.DataType record list. Only written for nonvirtual blocks and if data type index is not 0 (i.e., real_T).
ComplexSignal	yes: Only written for nonvirtual blocks and if signal is complex.
SignalSrc	Vector of length outputPortWidth giving the location of the signal source.
}	
NumBlocks	Number of nonvirtual blocks in the subsystem.
BlockSysIdx	System index for blocks in this subsystem.
BlockMap	Vector of length NumBlocks giving the blockIdx for each nonvirtual block in the subsystem.
}	

External Inputs and Outputs

The `model.rtw` file contains all information describing the external inputs (which correspond to root-level inport blocks) and external outputs (which correspond to root-level outport blocks). In control theory, the external inputs vector is conventionally referred to as U and the external output vector is referred to as Y . The generated code uses `rtU` and `rtY`.

Table A-9: Model.rtw External Inputs and Outputs

Variable/Record Name	Description
ExternalInputDefaults {	
DataTypeIdx	0: The default signal data type is <code>real_T</code> .
ComplexSignal	no: The default signal is not complex.
DirectFeedThrough	yes: The default assumes the root inport requires its input.
StorageClass	Auto: The default value specifies that the Real-Time Workshop decides how external signals are declared.
StorageTypeQualifier	"": The default type qualifier is empty.
}	
NumExternalInputs	Integer number of records that follow, one per root-level inport block
ExternalInput {	One record for each external input signal (i.e., root inport).
Identifier	Unique name across all external inputs.
TID	Integer task id (sample time index) giving the <code>SampleTime</code> record for this inport block.
SigIdx	[externalInputVectorIndex, signalWidth].
Dimensions	Vector of the form [nRows, nCols] for the signal. Only written if number of dimensions is greater than 1.
DataTypeIdx	Integer index of <code>DataType</code> record corresponding to this block. Only written if index is not 0.

Table A-9: Model.rtw External Inputs and Outputs (Continued)

Compl exSi gnal	yes: Only written if this inport signal is complex.
Si gLabel	Signal label entered by user.
Di rectFeedThrough	Only written if this inport doesn't require its signal when Mdl Outputs is called.
StorageCl ass	Only written if not Auto. This setting determines how this signal is declared.
StorageTypeQual i fi er	Only written if not empty.
}	
}	
External InputsMap	Matrix of dimension (NumModel Inputs, 2) , which gives a mapping from external input vector index (Ui) into the External Inputs structure: [<i>external InputsIndex</i> , <i>signal Offset</i>]. Only written if NumModel Inputs > 0.
External Outputs {	External outputs (root outputs) from the block diagram.
NumExternal Outputs	Number of External Output records that follow. This is equal to the number of root level outputs.
External Output {	One record per root-level output block.
Bl ock	[<i>sysIdx</i> , <i>bl ockIdx</i>] of the output block.
Si gIdx	[<i>external OutputVectorIndex</i> , <i>si gnal Wi dth</i>].
Di mensi ons	Vector of the form [<i>nRows</i> , <i>nCols</i>] for the signal. Only written if number of dimensions is greater than 1.
Si gLabel	Label on the input port signal, if any.
}	
}	

Data Store Information

The *model.rtw* records containing data store information include defaults for the data types and an indication whether or not the data is complex. The primary purpose of the DataStore record is to set up memory for implementing the data store. Each instance of a Data Store Read block that uses the same data store variable name is allowed to read the memory location(s) while Data Store Write blocks are allowed to write to the designated memory location(s). Data Store Read and Data Store Write blocks are placed under the block instance section of the *model.rtw* file.

Table A-10: Model.rtw Data Store Information

Variable/Record Name	Description
DataStoreDefaults {	Defaults for the data store records.
DataTypeIdx	0: Default is real_T.
ComplexSignal	no: Default is not complex.
}	
DataStores {	Record giving the data stores found in the block diagram.
NumDataStores	Number of data stores in the block diagram.
DataStore {	One record for each data store.
Name	Name of block declaring the data store.
SLName	Unmodified Simulink name. This is only written if it is <i>not</i> equal to Name.
MemoryName	Name of the data store memory region.
Identifier	Unique identifier across all data stores.
Index	[<i>dataStoreIndex</i> , <i>dataStoreWidth</i>].
Dimensions	Vector of the form [<i>nRows</i> , <i>nCols</i>] for the signal. Only written if number of dimensions is greater than 1.
InitValue	Initial value for the data store.
}	
}	

Block I/O Information

The block I/O vector (also referred to as the rtB vector) is described in the following BlockOutputs record. Each nonvirtual block output defines an entry in this conceptual vector. This record differs from the CompiledModel.RootSignals and CompiledModel.Subsystem records that describe the signal information for virtual and nonvirtual blocks. These two records also include model hierarchy information while the BlockOutputs record does not.

The BlockOutputs record provides a listing of all blocks that write to the block output vector. Several optimizations that affect block outputs are provided through Simulink dialog boxes. The **Advanced** page of the **Simulation Parameters** dialog box page provides the Signal Storage Reuse optimization. When you enable this option, rtwgen will attempt to reuse signal storage, mapping multiple BlockOutput records together. If you disable this option, rtwgen will create a unique record for all block signals. When this options is enabled, you can selectively add the output from a particular block by specifying the block output as a test point.

To specify a block output as a test point, select a line and then select **Edit -> Signal Properties** -> check box **SimulinkGlobal (Test Point)**. Once you have tagged a signal as a test point, the Target Language Compiler always writes to the block I/O vector. If a signal is not visible in the block outputs vector, it will allow reuse of its memory location by several blocks. This can substantially reduce memory requirements.

Table A-11: Model.rtw Block I/O Information

Variable/Record Name	Description
BlockOutputs {	List of block output signals in the block diagram.
BlockOutputDefaults {	
TestPoint	no: The default is that this signal has not been marked as a signal of interest in your model (see signal properties dialog).
StorageClass	Auto: The default value specifies that Real-Time Workshop decides where to declare this signal.
StorageTypeQualifier	"": The default type qualifier is empty.

Table A-11: Model.rtw Block I/O Information (Continued)

<code>IdentiferScope</code>	"top-level": The default is to declare the block output signal in the global block I/O vector.
<code>Invariant</code>	no: The default is that this signal has a non-constant sample time and change during execution.
<code>Initial Value</code>	[]: The default initial value is empty for non-invariant signals.
<code>DataTypeIdx</code>	0: The default data type is <code>real_T</code> .
<code>ComplexSignal</code>	no: The default is a non-complex real valued signal.
<code>SignalSrc</code>	[]: The default source is nonexistent for case of multiple sources that is created via reused signals.
<code>SignalLabel</code>	"": No signal label on the line.
<code>SignalConnected</code>	all: All destination elements of the signal are connected to other nonvirtual blocks or root outputs.
}	
<code>ReusedBlockOutputDefaults</code> {	
<code>SignalLabel</code>	"": No signal label on the line.
<code>SignalConnected</code>	all: All destination elements of signal are connected to other nonvirtual blocks or root outputs.
}	
<code>MergedBlockOutputDefaults</code> {	
<code>SignalLabel</code>	"": No signal label on the line.
<code>SignalConnected</code>	all: All destination elements of signal are connected to other nonvirtual blocks or root outputs.
}	
<code>NumBlockOutputs</code>	Number of data output port signals.
<code>BlockOutput</code> {	One record for each data output signal.

Table A-11: Model.rtw Block I/O Information (Continued)

Identifier	Unique variable name across all block outputs.
SignalIdx	[<i>blockIOVectorIndex</i> , <i>signalWidth</i>].
TestPoint	yes. Only written when this signal has been marked as a test point in the block diagram. Test point block outputs are always in the global scope ("top-level").
StorageClass	Only written if either "ExportedGlobal", "ImportedExtern" or "ImportedExternPointer". This setting determines how this signal is declared.
StorageTypeQualifier	Only written if non-empty (e.g., "const" or something similar).
IdentifierScope	"fcn-level": Only written when the output signal is local to a function. The default (above) is "top-level".
Invariant	yes: Only written when this block output cannot change during execution. For example, the output of a Width block and the output of a Constant block is invariant if <i>InlineParameters</i> =1.
InitialValue	Non-empty vector that is only written when <i>Invariant</i> is yes and the data type of the block output signal is a built-in data type.
DataTypeIdx	Only written when data is non-real_T (i.e., non-zero). This is the index in to the data type table that identifies this signals data type.
ComplexSignal	yes: Only written if this signal is complex.
SignalSrc	[<i>systemIndex</i> , <i>blockIndex</i> , <i>outputPortIndex</i>].
SignalLabel	Signal label entered by user. Only written if non-empty ("").
SignalConnected	Only written if one or more elements are not connected to destination non-virtual or root output blocks. In this case it will be none if no elements are connected or a vector of length <i>signalWidth</i> where each element is either a 1 or 0 indicating whether or not the corresponding output signal is connected.
NumMergedBlockOutputs	Number of MergedBlockOutput records. These occur when Merge blocks exist in your model. The number of these records will equal the number of merge block outputs in your model.

Table A-11: Model.rtw Block I/O Information (Continued)

<code>MergedBlockOutput {</code>	Only written if the <code>BlockOutput</code> record corresponds to a Merge block. In this case, the number of <code>MergedBlockOutput</code> records is equal to the number of input ports on the Merge block.
<code> Identifier</code>	Unique variable name across all block outputs.
<code> SigSrc</code>	[<i>systemIndex</i> , <i>blockIndex</i> , <i>outputPortIndex</i>].
<code> SigLabel</code>	Signal label entered by user. Only written if nonempty ("").
<code>}</code>	
<code>NumReusedBlockOutputs</code>	Number of <code>ReusedBlockOutput</code> records.
<code>ReusedBlockOutput {</code>	Only written when this <code>BlockOutput</code> record is being reused by multiple blocks. There is one record for each block output port that is reused by this <code>BlockOutput</code> record.
<code> Identifier</code>	Unique variable name across all block outputs.
<code> SigSrc</code>	[<i>systemIndex</i> , <i>blockIndex</i> , <i>outputPortIndex</i>].
<code> SigLabel</code>	Signal label entered by user. Only written if non-empty ("").
<code> MergedBlockOutput {</code>	Only written if this <code>ReusedBlockOutput</code> record corresponds to a Merge block. In this case, the number of <code>MergedBlockOutput</code> records is equal to the number of input ports on the Merge block. See above for contents of the <code>MergedBlockOutput</code> records.
<code> }</code>	
<code> }</code>	
<code>ObjectProperties {</code>	Only written if a Unified Data Repository object is attached to the output signal and the output signal is not being reused.

Table A-11: Model.rtw Block I/O Information (Continued)

...	Fields in the object properties record depend upon the contents of the object.
}	
}	
}	
BlockOutputsMap	Matrix of dimension (NumBlockSignals, 2), which gives a mapping from a block I/O vector index (Bi) into the BlockOutputs structure: [<i>blockOutputsIndex</i> , <i>signalOffset</i>]. Only written if NumBlockSignals > 0.

Data Type Work (DWork) Information

Certain blocks require persistence to store values between consecutive time intervals. When these blocks require the data to be stored in a data type other than `real_T` (the default data type), then instead of using an `RWork` element, a `DWork` element is used. `DWork` contains block identifier, name, width, datatype index, and a flag that tells whether it is used to retain data typed state information. Blocks that use data types but do not require persistence (e.g., Gain blocks) do not require `DWork` entries.

Note, all `RWork`, `IWork`, `PWork`, `Mode`, `DiscState` code elements are captured in the `DWork` records. Think of the `real` (`RWork`), `integer` (`IWork`), and `pointer` (`PWork`), etc. as well-defined data type (`DWork`) vectors.

Table A-12: Model.rtw Data Type (DWork) Information

Variable/Record Name	Description
<code>DWorkRecords {</code>	List of all data type work vectors in the model. There is one <code>DWorkRecord</code> record for each data type work vector in the model. The <i>source</i> of a data type work vector is a block. A block can have zero or more data type work vectors.
<code>DWorkRecordDefaults {</code>	Default values for the following <code>DWorkRecord</code> records.
<code>DataTypeIdx</code>	0: Default vector of a <code>DWorkRecord</code> record is a <code>real_T</code> vector.
<code>ComplexSignal</code>	no: Default vector of a <code>DWorkRecord</code> record is a non-complex vector.
<code>UsedAsDState</code>	no: Default vector of a <code>DWorkRecord</code> record is not logged as a state (i.e., doesn't go in to the <i>model.mat</i> file).
<code>}</code>	
<code>NumDWorks</code>	Number of data type work vectors in the model. Each block can register 0 or more data type work vectors. This include discrete states, <code>RWork</code> , <code>IWork</code> , <code>PWork</code> and <code>Mode</code> . Each record contains a vector as well as information describing the vector. For example, the model may contain two data type work records where one record contains a vector of length 3 and the other contains a vector of length 9 where each vector is of a different data type. In this case, <code>NumDWorks</code> is 2.

Table A-12: Model.rtw Data Type (DWork) Information (Continued)

Variable/Record Name	Description
DWork {	One DWork record for each data type work vector.
Identifier	Identifier provides a unique variable name across all data type work vectors.
Width	Length of the data type work vector.
DataTypeIdx	Index into the CompiledModel . DataTypes. DataType record list (i.e., the data type table used to identify the data type for this DWork). Only written if data type is a non-real_T (i.e., not a 0).
ComplexSignal	yes: Only written if this data type work vector is complex.
UsedAs	Only written if not default ("DWORK"), it can be either "MODE" or "RWORK" or "IWORK" or "PWORK" or "DSTATE".
SigSrc	[systemIdx, blockIdx, dworkIdx].
}	
}	

State Mapping Information

All continuous and discrete states contained within your model are conceptually grouped into a single vector, referred to as X (and rtX in the generated code). Blocks can directly connect to the state vector by using state ports. The `StatesMap` provides a mapping for these types of connections. The format of the `StatesMap` may change in a future release.

Table A-13: Model.rtw State Mapping Information

Variable/Record Name	Description
<code>StatesMap</code>	Matrix of dimension $(N, 3)$, where N = total number of state ports in your model. Block signal sources that come from a state vector (e.g., <code>Signal Src X4</code>) use the <code>StatesMap</code> to locate their source block. For example, “ <code>Signal Src X4</code> ” means the fifth row of the <code>StatesMap</code> , not the fifth index into the conceptual model state vector.

Block Record Defaults

When a block record does not contain an entry for a particular field located in the BlockDefault s record, then the BlockDefault s entry is used for the undeclared field.

Table A-14: Model.rtw Block Defaults

Variable/Record Name	Description
BlockDefault s {	Record for default values of block variables that aren't explicitly written in the block records. The block records only contain nondefault values for the following variables.
InMask	no
AlgebraicLoopId	0
PortBasedSampleTimes	no
ContStates	[0, 0]
ModeVector	[0, - 1]
RWork	[0, - 1]
IWork	[0, - 1]
PWork	[0, - 1]
DiscStates	[0, - 1]
NumDWork	0
NonsampledZCs	[0, 0]
ZCEvents	[0, 0]
RollRegions	[]
NumDataInputPorts	0
NumControlInputPorts	0
NumDataOutputPorts	0
Parameters	[0, 0]
}	

Parameter Record Defaults

The `ParameterDefaults` record contains default entries for parameters. These records are used throughout the model when no field is explicitly provided for a model parameter. For example, the default `DataTypeIdx` is 0, which corresponds to `real_T`. The default entry for `ComplexSignals` is `no`. Other entries such as the `Tunable` field is controlled by the **Inline parameters** check box. If for a given block instance, a `Parameter` record does not contain a specific entry for these fields, then the value from the `ParameterDefaults` is applied.

Table A-15: Model.rtw Parameter Defaults

Variable/Record Name	Description
<code>ParameterDefaults {</code>	Record for default values of block variables that aren't explicitly written in the block parameter records. The block parameter records only contain nondefault values for the following variables.
<code>DataTypeIdx</code>	0 (this corresponds to <code>real_T</code>)
<code>ComplexSignal</code>	<code>no</code>
<code>Tunable</code>	<code>off</code> : If inline parameters check box is off, otherwise <code>on</code> if inline parameters check box is on.
<code>StorageClass</code>	<code>Auto</code>
<code>}</code>	

Data and Control Port Defaults

In the event that `DataInputPort`, `ControlInputPort`, or `DataOutputPort` values are not provided in a block data record, then the default values are used as provided by these records. This includes information for data type index, complex signals, direct feed through, and a value for buffer destination ports (e.g., indicator for buffer reuse).

Table A-16: Model.rtw Data and Control Input Port Defaults

Variable/Record Name	Description
<code>DataInputPortDefaults {</code>	Record for default values of block variables that aren't explicitly written in the block data input port records. The block data input port records only contain nondefault values for the following variables.
<code>DataTypeIdx</code>	0
<code>ComplexSignal</code>	no
<code>FrameData</code>	no
<code>HaveGround</code>	no
<code>SrcHasImportedExternPointer</code>	no
<code>DirectFeedThrough</code>	yes. Only written if the <code>rtwgen</code> option <code>WriteBlockConnections</code> has been specified as on.
<code>BufferDstPort</code>	- 1: Default is no output ports are reusing the corresponding input port buffer.
<code>}</code>	
<code>ControlInputPortDefaults {</code>	Record for default values of block variables that aren't explicitly written in the block control (enable/trigger) input port records. The block control input port records only contain nondefault values for the following variables.
<code>DataTypeIdx</code>	0
<code>ComplexSignal</code>	no

Table A-16: Model.rtw Data and Control Input Port Defaults (Continued)

Variable/Record Name	Description
DirectFeedThrough	yes. Only written if the rtwgen option WriteBlockConnections has been specified as on.
FrameData	no
HaveGround	no
SrcHasImportedExternPointer	no
BufferDstPort	- 1: Default is no output ports are reusing the corresponding input port buffer.
}	
DataOutputPortDefaults {	Record for default values of block variables that aren't explicitly written in the block data output port records. The block data output port records only contain nondefault values for the following variables.
FrameData	no
Offset	- 1
Width	- 1
Dimensions	[- 1, - 1]
}	

Model Parameters Record

The model parameters record provides a complete description of the block parameters found within the model. The `CompiledModel.System[i].Block[i].Parameter[i].ASTNode` index into the `CompiledModel.ModelParameters.Parameter[i]` record.

Table A-17: Model.rtw Model Parameters Record

Variable/Record Name	Description
ModelParameters {	
NumParameters	Total number of unique parameter values (sum of next 5 fields).
NumInrtP	Number of parameter values in "rtP" parameter vector (realized as a struct). These are visible to external mode and possibly shared by multiple blocks.
NumInlinedUnlessRolled	Number of inlined parameter values. These are inlined, unless the roll threshold causes them to be placed in global memory. These parameters are not shared by multiple blocks.
NumExportedGlobal	Number of exported global parameters values. May be shared by multiple blocks.
NumImportedExtern	Number of imported parameter values. May be shared by multiple blocks.
NumImportedExternPointer	Number of parameter values that are accessed via imported extern pointers. May be shared by multiple blocks.
ParameterDefaults {	Default values for the following Parameter records.
DataTypeIdx	0: Default is real_T data type.
ComplexSignal	no: Default is non-complex
Tunable	no: Default value is not tunable.
StorageClass	Auto: Default value is Auto (Real-Time Workshop declares the memory).
TypeQualifier	"": Default is no type qualifier.

Table A-17: Model.rtw Model Parameters Record (Continued)

Variable/Record Name	Description
<code>IsSfcnSizePrm</code>	0: Default is not an S-function sizes parameter (only used by non-inlined S-functions)
<code>}</code>	
<code>Parameter {</code>	
<code>Identifier</code>	Identifier used in the generated code.
<code>Tunable</code>	If inlined parameters check box is off, then all parameter values are tunable (they will reside in the <code>rtP</code> vector). If inlined is on, then tunable means that this parameter has been selectively non-inlined. It will be placed in memory according to the Storage class. Note that the default value is 'no' (in which case this field is not written).
<code>RequiredInP</code>	Parameter required in <code>rtP</code> vector, 1 for the Accelerator, Rapid Simulation Target, External Mode; 0 otherwise.
<code>StorageClass</code>	Specifies where to declare/place this parameter value in memory (Auto, ExportedGlobal, ImportedExtern, ImportedExternPointer). Default value is Auto in which case this field is not written to the <code>model.rtw</code> file.
<code>TypeQualifier</code>	String used as a type qualifier for the declaration of the parameter (e.g., "static").
<code>Value</code>	Evaluated value of this parameter.
<code>Dimensions</code>	Actual dimensions of this parameter value. Note, it is possible for blocks to have matrix values written as a column-major vector. This field contains the dimensions of the data prior to the flattening of the vector to column-major.
<code>DataTypeIDx</code>	Data type index into the data type table (<code>CompiledModel.DataTypes.DataType</code>).

Table A-17: Model.rtw Model Parameters Record (Continued)

Variable/Record Name	Description
ComplexSignal	yes or no, is this a complex signal?
IsSfcnSizePrm	1 if this is an S-function sizes parameter, 0 otherwise.
ReferencedBy	An N-by-3 matrix. Each row specifies a system, block, parameter index triplet that identifies a usage of this parameter value. If N>1, then this parameter value is shared by multiple blocks.
}	
}	

System Record

The `System` record describes how to execute the blocks within your model. In general, a model can consist of multiple systems. There is one system for the root and one for each nonvirtual (conditionally executed) subsystem. All virtual (nonconditional) subsystems are *flattened* and placed within the current system. Each descendent system of the root system is written out using Pascal ordering (deepest first) to avoid forward references. Within each system is a sorted list of blocks.

Table A-18: Model.rtw System Record

Variable/Record Name	Description
<code>System {</code>	One for each system in the model. This is equal to <code>NumNonvirtSubsystems</code> plus 1 for the root system.
Type	root, atomic, enable, trigger, enable_with_trigger, or function-call.
Name	Name of system.
SLName	Unmodified Simulink name. This is only written if it is <i>not</i> equal to Name.
NoCode	If yes, generate no code. This system runs on the host only (during simulations or during external mode).
Identifier	Unique identifier across all blocks.
SystemIdx	System index assigned to this system. Each system is assigned a unique non-negative integer by <code>rtwgen</code> .
SubsystemBlockIdx	[<i>systemIndex</i> , <i>blockIndex</i>]. Not present if Type is root. Vector of two elements. First element is the index of the parent system of this system. The second element is the index of this system in the block list of its parent.
NumChildrenSystems	Number of systems that this system parents.
Children	Subsystem indices of the children. Note that the Children field is only written out if NumChildrenSystems is greater than 0.
ForceNonInline	Only relevant if you have a function-call subsystem. Otherwise it will always be off. Field is only written out for non-root systems.
NumZCEvents	Number of zero-crossing events for all blocks in this system.

Table A-18: Model.rtw System Record (Continued)

Variable/Record Name	Description
<code>InlineSubsystem</code>	Flag indicating whether subsystem should be inlined or not; Overridden by <code>ForceNonInline</code> flag for function-call subsystems if <code>InlineSubsystem</code> is on and <code>ForceNonInline</code> is on. Only written for non-root systems.
<code>UseSystemNameForRTWFileName</code>	Flag indicating whether the system name should be used for the code generation file name also. Only written for non-root systems.
<code>SystemFileName</code>	File name that code should be generated in to. Simulink determines the appropriate filename based on <code>UseSystemNameForRTWFileName</code> etc. so that TLC can directly use this file name. Only written for non-root systems.
<code>LibraryName</code>	Name of library that this system originated from if this block is a library link. This field is written out ONLY if the system is a library link or a descendent of it. Only written for non-root systems.
<code>StartFcn</code>	Name of start functions for nonvirtual subsystem.
<code>InitializeFcn</code>	Name of initialize function for enable systems that are configured to reset states.
<code>OutputFcn</code>	Name of output function for nonvirtual subsystem
<code>UpdateFcn</code>	Name of update function for nonvirtual subsystem.
<code>DerivativeFcn</code>	Name of derivative function for systems that have continuous states.
<code>EnableFcn</code>	Name of disable function for enable or enable_with_trigger systems.
<code>DisableFcn</code>	Name of disable function for enable or enable_with_trigger systems.
<code>ZeroCrossingFcn</code>	Name of nonsampled zero-crossing function for enable systems using variable step solver.
<code>OutputUpdateFcn</code>	Name of output/update function for trigger or enable_with_trigger systems.
<code>NumBlocks</code>	Number of nonvirtual blocks in the system.

Table A-18: Model.rtw System Record (Continued)

Variable/Record Name	Description
BlockIdx	0: Location of the first nonvirtual block in the following Block record list.
NumVirtualOutputBlocks	For the root system, the number of virtual output blocks is 0 (since all root output blocks are nonvirtual). For a system corresponding to a conditionally executed subsystem, this is equal to the number of output blocks in the subsystem. For each of these virtual output blocks, there is a corresponding Block record which appears after all the nonvirtual Block records.
VirtualOutputBlockIdx	Starting index in the following Block record list of the virtual output blocks.
NumTotalBlocks	Number of blocks in the system (sum of NumBlocks and NumVirtualOutputBlocks).
Block {	One for each nonvirtual block in the system. The virtual output block records are described below.
Type	Block type, e.g., Gain.
InMask	Yes if this block <i>lives</i> within a mask.
MaskType	Only written out if block is masked. If this property is yes, this block is either masked or resides in a masked subsystem. The default for MaskType is no meaning the block does not have a mask or resides in a masked subsystem.
Tag	This is the text that can be attached to a block via the command: <code>set_param('block', 'Tag', 'text')</code> This parameter is written if the text is non-empty.
RTWdata {	The RTWdata general record is only written if the RTWdata property of a block is non-empty. The RTWdata is created using the command: <code>set_param('block', 'RTWdata', val)</code> where <i>val</i> is a MATLAB struct of string. For example, <code>val.field1 = 'field1 value'</code> <code>val.field2 = 'field2 value'</code>
field1	"field1 value"
field2	"field2 value"

Table A-18: Model.rtw System Record (Continued)

Variable/Record Name	Description
}	
Name	Block name preceded with a <root> or <S#> token.
SLName	Unmodified Simulink name. This is only written if it is <i>not</i> equal to Name.
Identifier	Unique identifier across all blocks in the model.
PortBasedSampleTimes	yes. Only written if block specified port based sample times.
InputPortTIDs	Only written if port sample time information is available.
OutputPortTIDs	Only written if port sample time information is available.
TID	Task ID, which can be one of: <ul style="list-style-type: none">• Integer ≥ 0, giving the index into the sample time table.• Vector of two or more elements indicating that this block has multiple sample times.• constant indicating that the block is constant and doesn't have a task ID.• triggered indicating that the block is triggered and doesn't have a task ID.• Subsystem indicating that this block is a conditionally executed subsystem and the TID transitions are to be handled by the corresponding system.
SubsystemTID	Only written if TID equals Subsystem. This is the actual value of the subsystem TID (i.e., integer, vector, constant, or triggered).
FundamentalTID	Only written for multirate or hybrid enabled subsystems. This gives the sample time as the greatest common divisor of all sample times in the system.
SampleTimeIdx	Actual sample time of block. Only written for zero order hold and unit delay blocks.

Table A-18: Model.rtw System Record (Continued)

Variable/Record Name	Description
AlgebraicLoopId	This ID identifies what algebraic loop this block is in. If this field is not present, the ID is 0 and the block is not part of an algebraic loop.
ContStates	Specified as [N, I] where N is number of continuous states and I is the index into the state vector, X. Not present if N==0.
ModeVector	Specified as [N, I] where N is the number of model vector elements and I is the index into the data type work vector record list. Not present if N==0.
RWork	Specified as [N, I] where N is the number of real-work vector elements and I is the index into the data type work vector record list. Not present if N==0.
IWork	Specified as [N, I] where N is the number of integer-work vector elements and I is the index into the data type work vector record list. Not present if N==0.
PWork	Specified as [N, I] where N is the number of pointer-work vector elements and I is the index into the data type work vector record list. Not present if N==0.
DiscreteStates	Specified as [N, I] where N is the number of discrete state vector elements and I is the index into the data type work vector record list. Not present if N==0.
NumDWork	Number of DWork records block has declared. There is one DWork record for each data type work vector of the block.
DWork {	One record for each data type work vector.
Name	Name of the data type work vector.
RecordIdx	Index of this record in the model wide CompiledModel.DWorkRecords.DWorkRecord list.
}	
NonsampledZCs	Specified as [N, I], where N is the number of nonsampled zero-crossings and I is the index into the nonsampledZCs and nonsampledZCdirs vectors.
NonsampledZC {	One record for each nonsampled zero-crossing.

Table A-18: Model.rtw System Record (Continued)

Variable/Record Name	Description
Index	Index of the block's zero-crossing.
Direction	Direction of zero-crossing: Falling, Any, Rising.
}	
ZCEvents	Specified as [N, I], where N is the number of zero-crossing events and I is the index into the <code>zcEvents</code> vector.
ZCEvent {	One record for each zero-crossing event.
Type	Type of zero-crossing: DiscontinuityAtZC, ContinuityAtZC, TriggeredDiscontinuityAtZC.
Direction	Direction of zero-crossing: Falling, Any, Rising.
}	
RollRegions	<code>RollRegions</code> is the contiguous regions defined by the inputs and <i>block width</i> . Block width is the overall width of a block after scalar expansion. <code>RollRegions</code> is provided for use by the <code>%roll</code> construct.
NumDataInputPorts	Number of data input ports. Only written if nonzero.
DataInputPort {	One record for each data input port.
Width	Length of the signal entering this input port.
Dimensions	Vector of the form [nRows, nCols] for the signal. Only written if number of dimensions is greater than 1.
DataTypeIdx	Index into the <code>CompiledModel.DataTypes</code> . <code>DataType</code> record list giving the data type of this port. Only written if not 0 (see <code>CompiledModel.DataInputPortDefaults.DataTypeIdx</code>).
ComplexSignal	Is this port complex? Only written if yes. The default from <code>CompiledModel.DataInputPortDefaults.ComplexSignal</code> is no.
FrameData	yes/no: Is this port frame-based?
HaveGround	yes/no: Is this port connected to ground?

Table A-18: Model.rtw System Record (Continued)

Variable/Record Name	Description
Signal Src	A vector of length Width where each element specifies the source signal. This is an index into the block I/O vector (Bi), an index into the state vector (Xi), an index into the external input vector (Ui), unconnected ground (G0), or FcnCall indicating the source is a function-call.
RollRegions	A vector (e.g., [1:5, 6:10, 11]) giving the contiguous regions for this data input port over which <i>for</i> loops can be used. This is always written for S-Function blocks, otherwise it is written only if it is different from the block RollRegions.
DirectFeedThrough	Does this input port have direct feedthrough? Only written if WriteBlockConnections is on and the value this port does not have direct feedthrough, in which case no is written.
BufferDstPort	Only written if this input port is used by an output port of this block. The default is CompiledModel.DataInputPortDefaults.BufferDstPort which is -1.
}	
NumControlInputPorts	Number of control (e.g., trigger or enable) input ports. Only written if nonzero.
ControlInputPort {	One record for control input port.
Type	Type of control port: enable, trigger, or function-call.
Width	Width (i.e. vector length) of the signal entering this input port.
Dimensions	Vector of the form [nRows, nCols] for the signal. Only written if number of dimensions is greater than 1.
DataTypeIdx	Index into the CompiledModel.DataTypes.DataType record list giving the data type of this port. Only written if not 0 (see CompiledModel.ControlInputPortDefaults.DataTypeIdx).
ComplexSignal	Is this port complex? Only written if yes. The default from CompiledModel.ControlInputPortDefaults.ComplexSignal is no.
HaveGround	yes/no: Is this port connected to ground?

Table A-18: Model.rtw System Record (Continued)

Variable/Record Name	Description
Signal Src	A vector of length Width where each element specifies the source signal. This is an index into the block I/O vector (Bi), an index into the state vector (Xi), an index into the external input vector (Ui), or unconnected ground (G0).
Signal SrcTID	Vector of length Width giving the TID as an integer index, trigger, or constant identifier for each signal entering this control port. This is the rate at which the signal is entering this port. If the subsystem block has a triggered sample time, then the signal source must be triggered.
NumUniqueTIDs	Only written for enabled systems. Number of unique TIDs on the enable port, needed since there is only a mode element for each unique TID.
SrcTID	For each unique TID, a record which contains the tid and the roll regions on the enable port for that tid.
RollRegions	A vector (e.g., [1:5, 6:10, 11]) giving the contiguous regions for this data input port over which <i>for</i> loops can be used. This is always written for S-Function blocks, otherwise it is written only if it is different from the block RollRegions.
DirectFeedThrough	Does this input port have direct feedthrough? Only written if WriteBlockConnections is on and the value this port does not have direct feedthrough, in which case no is written.
BufferDstPort	Only written if this input port is used by an output port of this block. The default is CompiledModel.ControlInputPortDefaults.BufferDstPort which is -1.
}	
NumDataOutputPorts	Number of output ports. Only written if nonzero.
DataOutputPort {	One record for each output port.
Index	Index in the BlockOutputs map.
Dimensions	Vector of the form [nRows, nCols] for the signal. Only written if number of dimensions is greater than 1.

Table A-18: Model.rtw System Record (Continued)

Variable/Record Name	Description
FrameData	yes, no, or mixed: Is this port frame-based?
Offset	The offset of this port in its BlockOutputs which can be non-zero due to the merge block.
Width	The width of this port which can be different than width of its BlockOutputs due to the merge block.
}	
Connections {	Only written if this is an S-Function block, or the WriteBlockConnections rtwgen option was specified as on.
InputPortContiguous	Vector of length NumDataInputPorts containing yes, no, or grounded.
DirectSrcConn	Vector of length NumDataInputPorts containing yes or no as to whether or not the input port is directly connected to a nonvirtual source block.
DirectDstConn	Vector of length NumDataOutputPorts containing yes or no as to whether or not the output port is directly connected to a signal nonvirtual destination block.
DataOutputPort {	One record for each data output port.
NumConnPoints	Number of destination <i>connection points</i> . A destination connection point is defined to be a one-to-one connection with elements from the output (src) port to the destination block and port.
ConnPoint {	
SrcSignal	Vector of length two giving the range of signal elements for the connection: [startIdx, length] Where startIdx is the starting index of the connection in the output port and length is the number of elements in the connection.

Table A-18: Model.rtw System Record (Continued)

Variable/Record Name	Description
<code>DstBlockAndPortEl</code>	Vector of length four giving the destination connection: [<i>sysIdx</i> , <i>blkIdx</i> , <i>inputPortIdx</i> , <i>inputPortEl</i>] <i>sysIdx</i> is the index of the system record. <i>blkIdx</i> is the index with in system record of the destination block. <i>inputPortIdx</i> is the index of the destination input port. <i>inputPortEl</i> is the starting offset within the port of the connection.
<code>}</code>	
<code>}</code>	
<code>}</code>	
<code>ParamSettings {</code>	Optional record specific to block.
<code> blockSpecificName</code>	Block specific settings.
<code>}</code>	
<code><S-function fields></code>	Optional fields (parameters and/or records) that are written to the <i>model.rtw</i> file by the your specific S-function mdl RTW method.
<code>Parameters</code>	Specified as [N, M] where N is the number of Parameter records that follow, M is the number of modifiable parameter elements. Not present if N==0.
<code>Parameter {</code>	One record for each parameter.
<code> Name</code>	Name of the parameter as defined by the block.
<code> Dimensions</code>	Vector of the form [<i>nRows</i> , <i>nCols</i>] for the signal. Only written if number of dimensions is greater than 1.
<code> DataTypeIdx</code>	Data type index of the parameter into the <code>CompiledModel.DataTypes.DataType</code> records. Only written if not 0 (i.e., not <code>real_T</code>).
<code> ComplexSignal</code>	Is this parameter complex? Only written if yes.
<code> String</code>	String entered in the Simulink block dialog box.

Table A-18: Model.rtw System Record (Continued)

Variable/Record Name	Description
StringType	One of: <ul style="list-style-type: none"> • "Computed" indicating the parameter is computed from values entered in the Simulink dialog box. • "Variable" indicating the parameter is derived from a single MATLAB variable. • "Expression" indicating the parameter is a MATLAB expression.
ASTNode {	Contains the direct mapping of this parameter to the model parameters record list. Essentially, this is the 'value' of the parameter.
Op	<ul style="list-style-type: none"> • Op = SL_CALCULATED => AstNode contains ModelParametersIdx, an index into the Model Parameters table for evaluated (calculated) parameter expressions, • Op = SL_NOT_INLINED => AstNode contains ModelParametersIdx, an index into the Model Parameters table for evaluated (calculated) parameter expressions, • Op = SL_INLINED => AstNode contains ModelParametersIdx, an index into the Model Parameters table for evaluated (calculated) parameter expressions, • Op = M_ID (a terminal node), the AST record contains ModelParameterIdx. • Op = M_NUMBER (a terminal node), the AST record contains the numerical value (Value field). • Op = Simulink name of operator token (many). In this case, the ASTNode contains the fields NumChildren and the records for the children.
<i>fields depend on Op</i>	
}	
}	
ParamName0	Parameter[0] - An alias to the first parameter.

Table A-18: Model.rtw System Record (Continued)

Variable/Record Name	Description
...	
ParamNameN-1	Parameter[N-1] - An alias to the last parameter.
}	
Block {	One block record (after the nonvirtual block records) for each virtual output block in the system.
Type	Outport
Name	Block name preceded with a <root> or <S#> token.
SLName	Unmodified Simulink name. This is only written if it is <i>not</i> equal to Name.
Identifier	Unique identifier across all blocks.
RollRegions	A vector (e.g., [1:5, 6:10, 11]) giving the contiguous regions over which <i>for</i> loops can be used.
NumDataInputPorts	1
DataInputPort {	See nonvirtual block DataInputPort record.
}	
}	
EmptySubsysInfo {	
NumRTWdatas	Number of empty subsystem blocks that have set_param(block, 'RTWdata', val) specified, where val is a struct of strings.

Table A-18: Model.rtw System Record (Continued)

Variable/Record Name	Description
RTWdata {	The RTWdata general record is only written if the RTWdata property of a block is non-empty. The RTWdata is created using the command: <code>set_param(' block' , ' RTWData' , val)</code> where val is a MATLAB struct of string. For example, <code>val . fi el d1 = ' fi el d1 val ue'</code> <code>val . fi el d2 = ' fi el d2 val ue'</code>
fi el d1	"fi el d1 val ue"
fi el d2	"fi el d2 val ue"
}	
}	

Stateflow Record

Stateflow library charts contained within your model can be multiinstanced, meaning that more than one instance of the same library chart appears in your model. The `Stateflow` record contains one `Chart` record for each unique library chart. Each `Chart` record contains the block references to the chart. This record is used by the Real-Time Workshop/Stateflow code generator tools to generate code that is reusable among all instances of a library chart. Note that when generating code that will use dynamic memory allocation (e.g., `grt_malloc`), all Stateflow charts are treated as multiinstanced to allow reuse of the chart code.

Table A-19: Model.rtw Stateflow Record

Variable/Record Name	Description
<code>SFLibraryNames {</code>	Only written if Stateflow charts exist in the model.
<code>NumUniqueCharts</code>	Number of Chart records.
<code>Chart {</code>	Record for a Stateflow library chart.
<code>Name</code>	Name of the Stateflow library chart.
<code>ReferencedBy</code>	An N-by-2 matrix. Each row specifies a system and block pair that identifies a instance of the library chart.
<code>}</code>	
<code>}</code>	

Model Checksums

Checksums are created that are unique for each model.

Table A-20: Model.rtw Checksums

Variable/Record Name	Description
BlockParamChecksum	This is a hash-based checksum for the block parameter values and identifier names.
Model Checksum	This is a hash-based checksum for the model structure.

Block Specific Records

Each block may have parameters. All parameters are written out to the *model.rtw* file in Parameter records that are contained with the Block records. There is one Parameter record for each block parameter (i.e., Block.Parameter[i]). A parameter in this context only refers to parameters that external mode can tune. Therefore, there may not be a one-to-one mapping between parameters in the *model.rtw* file and the parameter dialog for the block.

Each Simulink built-in block has an associated block record that covers all possible configurations of that block. The following table provides a complete listing of built-in blocks and their associated records. The blocks are listed in alphabetical order. The Target Language Compiler also has an associated TLC file for each block that specifies how code is generated for that block.

This table describes the block specific records written for the Simulink blocks (excluding common fields described above).

Table A-21: Model.rtw Block Specific Records

Block Type	Properties
AbsoluteValue	No block specific records
Actuator	No block specific records.
Backlash	<ul style="list-style-type: none">BacklashWidth parameter giving the 'backlash' region for the block.1 RWorkDefine record, containing PrevY if fixed-step solver.2 RWorkDefine records, containing PrevYA and PrevYB used for 'banking' the output to prevent model execution inconsistencies.
BusSelector	No block specific records.
Clock	No block specific records.
CombinatorialLogic	TruthTable parameter defining what the output should be, $y = f(\text{TruthTable}, u)$.
ComplexToMagnitudeAngle	Output ParamSetting. Output is "Magnitude", "Angle", or "MagnitudeAndAngle" indicating what the output port(s) are producing.
ComplexToRealImag	Output ParamSetting. Output is one of "Real", "Imag", or "RealAndImag" indicating what the output port(s) are producing.
Constant	Value parameter indicating what the output port should produce.
DataStoreMemory	Virtual - Not written to the <i>model.rtw</i> file.

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
DataStoreRead	DataStore parameter - Region index into data stores list to get data store name, etc.
DataStoreWrite	DataStore parameter - Region index into data stores list to get data store name, etc.
DataTypeConversion	No block specific records.
DeadZone	<ul style="list-style-type: none"> LowerValue parameter - the lower value of the deadzone. UpperValue parameter - the upper value of the deadzone. InputContiguousParamSetting (yes or no). SaturateOnOverflow ParamSetting (NotNeed, Needed, NeededBugOff, or NeededForDiagnostics).
Demux	Virtual - Not written to the <i>model.rtw</i> file.
Derivative	<p>The Derivative block computes its derivative by using the approximation:</p> $(input - prevInput) / \Delta T$ <p>Two banks of history are needed to keep track of the previous input because the input history is updated prior to integrating states. To guarantee correctness when the output of the Derivative block is integrated directly or indirectly, two banks of the previous inputs are needed. This history is saved in the real-work vector (RWork). The real-work vectors are:</p> <ul style="list-style-type: none"> TimeStampA RWork - time values for 'bank A' LastUAtTimeA RWork - last input value for 'bank A' TimeStampB RWork - time values for 'bank B' LastUAtTimeB RWork - last input value for 'bank B'
DigitalClock	No block specific records.
DiscreteFilter	See Model.rtw Linear Block Specific Records.

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
DiscreteIntegrator	<ul style="list-style-type: none"> • Zero, one or two RWork vectors depending on the IntegratorMethod. These will be SystemEnable or ICNeedsLoading or both. • IntegratorMethod ParamSetting - ForwardEuler, BackwardEuler, or Trapezoidal. • ExternalReset ParamSetting - none, rising, falling, either, level. • InitialConditionSource ParamSetting - internal or external. • LimitOutput ParamSetting - on or off. • ShowSaturationPort ParamSetting - on or off. • ShowStatePort ParamSetting - on or off. • ExternalX0 ParamSetting - only written when initial condition (IC) source is external. This is the initial value of the signal entering the IC port. • InitialCondition parameter. • UpperSaturationLimit parameter. • LowerSaturationLimit parameter.
DiscretePulseGenerator	<ul style="list-style-type: none"> • PhaseDelay ParamSetting, giving the numerical phase delay. • OneIWorkforClockTicksCounter, used to manage the pulse. • Amplitude parameter, a numerical vector giving the pulse amplitude. • Period parameter, a numerical vector giving the pulse period. • PulseWidth parameter, a numerical vector giving the pulse width.
DiscreteStateSpace	See Model.rtw Linear Block Specific Records.
DiscreteTransferFcn	See Model.rtw Linear Block Specific Records.
DiscreteZeroPole	See Model.rtw Linear Block Specific Records.
Display	No block specific records.
ElementaryMath	Operator ParamSetting - One of sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh, exp, log, log10, floor, ceil, sqrt, reciprocal, pow, or hypot.

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
EnablePort	<p>Only written if nonvirtual. When nonvirtual, we write the following:</p> <ul style="list-style-type: none"> ControlPortNumber ParamSetting - The control input port number for this block. The corresponding subsystem block control input port index is the block port number minus one. SubsystemIdx ParamSetting - This is the location [systemIdx, blockIdx] of the nonvirtual subsystem which contains this nonvirtual Enable block.
From	Virtual. Not written to <i>model.rtw</i> file.
FromFile	<ul style="list-style-type: none"> FileName ParamSetting - Name of MAT-file to read data from NumPoints ParamSetting - Number of points of data to read TUData ParamSetting - Time and data points. Not present if using the Rapid Simulation Target. Width ParamSetting - Number of columns in TUData structure. OnePWork vector, PrevTimePtr used for managing the block output.

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
FromWorkspace	<ul style="list-style-type: none">• VariableName ParamSetting - Name of variable in “Data” field in block parameter dialog box.• DataFormat ParamSetting - "Matrix" or "Structure".• Interpolate ParamSetting - Interpolate flag is on/off (see entry for From Workspace block in the <i>Using Simulink</i> manual).• OutputAfterFinalValue ParamSetting - How to generate output after final data value (see entry for From Workspace block in the <i>Using Simulink</i> manual).• NumPoints ParamSetting - Number of data points (rows) over which to read values from as time moves forward and write to the output port. <p>The following two items (Time and Data) are written as Parameters if we are generating code for the Rapid Simulation Target, otherwise they are written as ParamSettings.</p> <ul style="list-style-type: none">• Time - The time tracking vector. May or may not be present. If data format is Matrix, then this field is always present. If data format is Struct then this field is present only if the time field exists.• Data - The data to put on the output port.• One IWork vector for the PrevIndex used in computing the output.• Three PWork vectors, TimePtr, DataPtr, RSInfoPtr used in computing the output.

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
Fcn	<ul style="list-style-type: none"> Expr ParamSetting - Text string containing the expression the user entered. ASTNode record, containing the parsed abstract syntax tree for the expression. The general form of the ASTNode is: <pre> ASTNode { Op Operator (e. g. "+") LHS { Left-hand side argument for Op ... } RHS { Right-hand side argument for Op ... } } </pre>
Gain	<ul style="list-style-type: none"> SaturateOnOverflow ParamSetting - Only written for element gain operations. (NotNeed, Needed, NeededBugOff, or NeededForDiagnostics). OperandComplexity ParamSetting - Only written for non-element gain operations. This is one of RR, RC, CR, CC where R=Real and C=Complex, depending on how the block is configured. Dimensions ParamSetting - Only written for non-element gain operations. This is a vector containing the dimensions for the gain operation. Complexities ParamSetting - Only written for non-element gain operations. An integer array [outputPortComplexity, <input and gain complexity pair>].
Goto	Virtual. Not written to <i>model.rtw</i> file.
GotoTagVisibility	Virtual. Not written to <i>model.rtw</i> file.
Ground	Virtual. Not written to <i>model.rtw</i> file.
HiddenBuffer	There are no block specific records. This block is automatically inserted into your model by the simulation engine to make the generate code more efficient by providing contiguous signals to blocks that require contiguous inputs (for example, the matrix multiply algorithm is more efficient if the inputs are contiguous).

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
HitCross	<ul style="list-style-type: none"> InputContiguous ParamSetting - yes, no is the input contiguous? HitCrossingOffset Parameter - The hit crossing offset used in computing the output.
Initial Condition	Value parameter - This is the initial condition to output the first time the block executes. It is a parameter (as opposed to a ParamSetting) to enable loop rolling.
Inport	Virtual. Not written to <i>model.rtw</i> file.
Integrator	<ul style="list-style-type: none"> External Reset ParamSetting - one of none, rising, falling, either, level. InitialConditionSource ParamSetting - internal or external. LimitOutput ParamSetting - on or off. ShowSaturationPort ParamSetting - on or off. ShowStatePort ParamSetting - on or off. External X0 ParamSetting - only present for external initial conditions. InputContiguous ParamSetting - is the first input port contiguous (yes or no)? ResetInputContiguous ParamSetting - Only present if the reset port is present. Initial Condition parameter. UpperSaturationLimit parameter. LowerSaturationLimit parameter.
Logic	Operator ParamSetting - one of AND, OR, NAND, NOR, XOR, or NOT.
Lookup	<ul style="list-style-type: none"> ZeroTechnique ParamSetting - The type of lookup being performed. This doesn't change during model execution. The possibilities are: Normal Interp, AverageValue, or MiddledValue. InputValues parameter - The input values, x, corresponding to the function $y = f(x)$. OutputValues parameter - The output values, y, of the function $y = f(x)$. OutputAtZero parameter - the output when the input is zero.

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
Lookup2D	<ul style="list-style-type: none"> ColZeroTechnique ParamSetting - Normal Interp, AverageValue, or MinValue. ColZeroIndex ParamSetting - Primary index when column data is zero. ColZeroIndex ParamSetting - Normal Interp, AverageValue, or MinValue. RowIndex parameter - The <i>row</i> input values, x, to the function $z = f(x, y)$. ColumnIndex parameter - The <i>column</i> input values, y, to the function $z = f(x, y)$. OutputValues parameter - The <i>table</i> output values, z, for the function $z = f(x, y)$.
MagnitudeAngleToComplex	<ul style="list-style-type: none"> Input ParamSetting - one of "Magnitude", "Angle", or "MagnitudeAndAngle" ConstantPart parameter - Only written when there is one input port.
Math	Operator ParamSetting - exp, log, 10^u , log10, square, sqrt, pow, reciprocal, hypot, rem, or mod.
MATLABFcn	There is no support for the MATLAB Fcn block in the Real-Time Workshop.
Memory	<ul style="list-style-type: none"> One DWork vector, PreviousInput, used to produce the output. X0 parameter - the initial condition.
Merge	Initial Output parameter, giving the initial output for the merged signal.
MinMax	Function ParamSetting - min or max.
MultiPortSwitch	No block specific records.
Mux	Virtual. Not written to <i>model.rtw</i> file.

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
Outport	<p>The block record for this block depends on the type of outport:</p> <ul style="list-style-type: none">• Root outports:<ul style="list-style-type: none">- PortNumber ParamSetting - Port number as entered in the dialog box.- OutputLocation ParamSetting - Specified as Yi if root-level outport; otherwise specified as Bi .- OutputWhenDisabled ParamSetting - Only written when in an enabled subsystem and will be held or reset.• Outport in a nonvirtual subsystem:<ul style="list-style-type: none">- InputContiguous ParamSetting - yes or no.- OutputWhenDisabled ParamSetting - held or reset.- SpecifyIC ParamSetting - yes or no was the IC specified?- InitialOutput parameter - Only written for virtual outport blocks in a nonvirtual subsystem.
Probe	<ul style="list-style-type: none">• ProbeWidth ParamSetting - on or off.• ProbeSampleTime ParamSetting - on or off.• ProbeComplexSignal ParamSetting - on or off.• ProbeSignalDimensions ParamSetting - on or off.
Product (element-wise multiply with one input port)	<ul style="list-style-type: none">• If block is configured for element-wise multiply, the block record contains:<ul style="list-style-type: none">- One optional IWork vector to suppress warnings.- Multiplication ParamSetting - "Element-wise(.*)"- Inputs ParamSetting - string vector of the form: ["*", "*", "/"]- SaturateOnOverflow ParamSetting (NotNeed, Needed, NeededBugOff, or NeededForDiagnostics).

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
Product (matrix multiply with one input port)	<div>If block is configured for matrix multiply with one input port, the block record contains:<ul style="list-style-type: none">• Multiplication ParamSetting - "Matrix(*)"• Inputs ParamSetting - string vector of the form: ["*", "*", "/"]• OneInputMultiply ParamSetting - yes.</div>

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
Product (matrix multiply with more than one input port).	<p>If block is configured for matrix multiply, with one input port, the block record contains:</p> <ul style="list-style-type: none">• <code>Multiplication ParamSetting</code> - "Matrix(*)"• <code>Inputs ParamSetting</code> - string vector of the form: ["*", "*", "/"]• <code>OneInputMultiply ParamSetting</code> - no.• <code>OperandComplexity ParamSetting</code> - RR, RC, CR, or CC where: RR : in1 (real) in2 (real) RC : in1 (real) in2 (complex) CR : in1 (complex) in2 (real) CC : in1 (complex) in2 (complex)• <code>Dimensions ParamSetting</code> - [numSteps x 3] matrix. Each row of the matrix contains 3 elements. If for a specific step, e.g., i-th, operand1 is a [m x n] matrix, and operand2 is a [n x k] matrix, the i-th row contains [m n k].• <code>Operands ParamSetting</code> - [numSteps x 3] matrix. Each row contains the {result, operand1, operand2}. Where: zero - block output greater than zero - data input port number (unity-index based) less than zero - dwork buffer number (negative unity-index based).• <code>Complexities ParamSetting</code> - [numSteps x 3] matrix. Each row contains the {result, operand1, operand2}. Where: zero - real one - complex• <code>Operators ParamSetting</code> - LU, Pivot, X dwork indices.• <code>DivisionBuffers ParamSetting</code> - LU, Pivot, X dwork indices.
Quantizer	<p><code>QuantizationInterval</code> parameter - numerical vector giving the quantization interval points.</p>

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
RandomNumber	<ul style="list-style-type: none"> • One IWork vector RandSeed. • One RWork vector NextOutput. • Mean parameter - the mean of the random number generator. • StandardDeviation parameter - the standard deviation of the random number generator.
RateLimiter	<ul style="list-style-type: none"> • If a variable step solver is being used, then this block has two RWork vectors, PrevYA and PrevYB (two banks to maintain consistent simulation results). • If a fixed-step solver is being used, then this block has two RWork vectors, PrevT and PrevY (used to keep track last time and output). • RisingSlewLimit parameter. • FallingSlewLimit parameter.
RealImagToComplex	<ul style="list-style-type: none"> • Input ParamSetting - Real, Imag, or RealAndImag. • ConstantPart parameter.
Reference	Will never appear in <i>model.rtw</i> .
RelationalOperator	<ul style="list-style-type: none"> • Operator ParamSetting - One of ==, ~=, <, <=, >=, >. • InputContiguous ParamSetting - yes or no.
Relay	<ul style="list-style-type: none"> • InputContiguous ParamSetting - yes or no. • OnSwitchValue parameter. • OffSwitchValue parameter. • OnOutputValue parameter. • OffOutputValue parameter.
ResetIntegrator	Initial Condition parameter.
Rounding	Operator ParamSetting - floor, ceil, round, or fix
Saturate	<ul style="list-style-type: none"> • InputContiguous ParamSetting - yes or no. • UpperLimit parameter. • LowerLimit parameter.

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
Scope	<ul style="list-style-type: none">• One PWork for LoggedData.• SaveToWorkspace ParamSetting - yes or no.• SaveName ParamSetting - name of variable to log.• MaxRows ParamSetting - maximum number of data points to log.• Decimation ParamSetting - integer giving when to log data 1 for every time step, 2 for every other time step, and so on.• DataFormat ParamSetting - StructureWithTime, Structure, or Matrix.• AxesTitles ParamSetting - record giving the axis title strings.• AxesLabels ParamSetting - record giving the axis label strings.• PlotStyles ParamSetting - what we are plotting.
Selector	Virtual. Not written to <i>model.rtw</i> file.

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
S-Function	<p>The S-function has the following parameter settings:</p> <ul style="list-style-type: none"> • FunctionName - Name of S-function. • SFunctionLevel - Level of the S-function 1 or 2. • FunctionType - Type of S-function: "M-File", "C-MEX", or "FORTRAN-MEX". • Inlined - yes, no, or skip. Skip is for case of non-C-MEX S-function sink. • DirectFeedthrough - For level 1 S-functions, this will be written as yes or no. For level 2 S-functions, this will be a vector of yes or no for each input port. • UsingUPtrs - If this is a Level 1 C MEX S-function and if it is using <code>ssGetUPtrs</code> (instead of <code>ssGetU</code>), then this <code>ParamSetting</code> will be "yes". If this a Level 2 S-function, then this field will be a vector of yes/no, each element corresponding to each input port. An element value of "yes" implies that the S-function has set the <code>RequiredContinuous</code> attribute for the corresponding input port to true. The Level 2 S-function will be using <code>ssGetInputPortSignal</code> (instead of <code>ssGetInputPortSignalPtrs</code>). • InputContinuous - For level 1 S-functions, this will be yes or no. For level 2 S-functions, this is a vector of yes or no for each input port. • SampleTimesToSet - Mx2 matrix of sample time indices indicating any sample times specified by the S-function in <code>mdlInitializeSizes</code> and <code>mdlInitializeSampleTimes</code> that get updated. The first column is the S-function sample time index, and the 2nd column is the corresponding <code>SampleTime</code> record of the model giving the <code>PeriodAndOffset</code>. For example, an inherited sample time will be assigned the appropriate sample time such as that of the driving block. In this case, the <code>SampleTimesToSet</code> will be [0, <i>] where <i> is the specific <code>SampleTime</code> record for the model.

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties												
S-Function (continued)	<ul style="list-style-type: none">• DynamicallySizedVectors - Vector containing any of: "U", "U0", "U1", ..., "Un", "Y", "Y0", "Y1", ..., "Yn", "Xc", "Xd", "RWork", "IWork", "PWork", "D0", ..., "Dn". For example ["U0", "U1", "Y0"]. For a level 1 S-function only U or Y will be used whereas for a level 2 S-function, U0, U1, ..., Un, Y0, Y1, ..., Yn will be used. This includes dynamically typed vectors, i.e., data type and complex signals. For example, if U0 is in this list either width, data type, or complex signal of U0 is dynamically sized (or typed).SFcnmdlRoutines - Vector containing any of: ["mdlInitializeSizes", "mdlInitializeSampleTimes", "mdlInitializeConditions", "mdlStart", "mdlOutputs", "mdlUpdate", "mdlDerivatives", "mdlTerminate" "mdlRTW"] Indicating which routines need to be executed. Only written for level 2 S-functions.• RTWGenerated - yes or no, is this generated by the Real-Time Workshop? <p>The next section contains information about function-call connections:</p> <table><tr><td>NumSFcnSysOutputCalls</td><td>- Number of calls to subsystems of type "function-call".</td></tr><tr><td>SFcnSystemOutputCall {</td><td>One record for each call</td></tr><tr><td> OutputElement</td><td>Index of the output element that is doing the function call.</td></tr><tr><td> FcnPortElement</td><td>Index of the subsystem function port element that is being <i>called</i>.</td></tr><tr><td> BlockToCall</td><td>[<i>systemIndex</i>, <i>blockIndex</i>] or unconnected</td></tr><tr><td>}</td><td></td></tr></table>	NumSFcnSysOutputCalls	- Number of calls to subsystems of type "function-call".	SFcnSystemOutputCall {	One record for each call	OutputElement	Index of the output element that is doing the function call.	FcnPortElement	Index of the subsystem function port element that is being <i>called</i> .	BlockToCall	[<i>systemIndex</i> , <i>blockIndex</i>] or unconnected	}	
NumSFcnSysOutputCalls	- Number of calls to subsystems of type "function-call".												
SFcnSystemOutputCall {	One record for each call												
OutputElement	Index of the output element that is doing the function call.												
FcnPortElement	Index of the subsystem function port element that is being <i>called</i> .												
BlockToCall	[<i>systemIndex</i> , <i>blockIndex</i>] or unconnected												
}													

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
S-function (continued)	<p>If the S-function has a mdl RTW method, then additional items can be added. See <i>matlabroot/simulink/src/sfuntmpl.doc</i>.</p> <p>If the S-function is not inlined, i.e., <i>sfunctiofname.tlc</i> does not exist) then</p> <p>For each S-function parameter entered in the dialog box, there is a P#Size and P# parameter giving the size and value of the parameter, where # is the index starting at 1 of the parameter in the dialog box.</p> <p>If the S-function is inlined, i.e., <i>sfunctiofname.tlc</i> does exist,</p> <p>No sizes parameter. Parameter names are derived from the run-time parameter names.</p>
Signal Generator	<ul style="list-style-type: none"> WaveForm ParamSetting - sine, square, or sawtooth. TwoPi - 6.283185307179586. Amplitude parameter. Frequency parameter.
Signum	No block specific records.
Sin	<p>This block has two very distinct forms. If the block is discrete, we use trigonometric identities to remove time from the output function, otherwise we simply compute the output as a function of time.</p> <ul style="list-style-type: none"> If the block is discrete, <ul style="list-style-type: none"> We have two RWork vectors, LastSin, LastCos and one IWork, SystemEnable. We have the following parameters: Amplitude, Frequency, SinH, CosH, SinPhi, CosPhi. otherwise: <ul style="list-style-type: none"> We have the following parameters: Amplitude, Frequency.
Step	<ul style="list-style-type: none"> Time parameter. Before parameter. After parameter.
StateSpace	See Model.rtw Linear Block Specific Records.

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
Sum	<ul style="list-style-type: none"> Inputs ParamSetting - A vector of the form ["+", "+", "-"] corresponding to the configuration of the block. SaturateOnOverflow ParamSetting (NotNeed, Needed, NeededBugOff, or NeededForDiagnostics).
SubSystem	<ul style="list-style-type: none"> SystemIdx ParamSetting - Index of this system in the <i>model.rtw</i> file. StatesWhenEnabling ParamSetting - held or reset. Only written if enable port is present. TriggerBlock ParamSetting - Block index of TriggerPort block in system or NotPresent. TriggerScope ParamSetting - Only written if we have a trigger port. It can be one of: NoScope, ScopeOnce, ScopeIndividually. EnableScope ParamSetting - Only written if we have an enable port. It can be one of: NoScope, ScopeOnce, ScopeIndividually. SystemContStates ParamSetting - Specified as [N, I] where N is the number of continuous states and I is the index into the state vector, X. UseSystemNameForRTWFileName ParamSetting - on or off. SystemFileName ParamSetting. NumNonsampledZCs ParamSetting. StartNonsampledZCs ParamSetting. SkipIntgOnTrigEvent ParamSetting. SingleRate ParamSetting. MinorStepGuard ParamSetting.
Switch	<ul style="list-style-type: none"> ControlInputContiguous ParamSetting - yes or no. Threshold parameter.
ToFile	<ul style="list-style-type: none"> One IWork vector, Decimation. Two PWork vectors, FilePtr, and LogFilePtr. Filename ParamSetting. MatrixName ParamSetting. Decimation ParamSetting.

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
ToWorkspace	<ul style="list-style-type: none"> • One PWork vector, LoggedData. • VariableName ParamSetting - Name of variable used to save data. • BufferParamSetting- Maximum number of rows to save or 0 for no limit. • DecimationParamSetting - Data logging interval • InputContinuous - yes or no. • SaveFormat ParamSetting. • Label ParamSetting.
Terminator	Virtual. Not written to <i>model.rtw</i> file.
TransferFcn	See Model.rtw Linear Block Specific Records.
TransportDelay	<ul style="list-style-type: none"> • InitialInput ParamSetting. • BufferSize ParamSetting. • DiscreteInput ParamSetting. • One IWork vector, BufferIndices. • One PWork vector, TUBuffer. • DelayTime parameter.
TriggerPort	<ul style="list-style-type: none"> • TriggerType ParamSetting - Only written if the number of output ports is one. • ControlPortNumber ParamSetting - The control input port number for this block. The corresponding subsystem block control input port index is PortNumber- 1. • SubsystemIdx ParamSetting - This is the location [systemIdx, blockIdx] of the non-virtual subsystem which contains this non-virtual Trigger block.
Trigonometry	<ul style="list-style-type: none"> • Operator ParamSetting - sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, or tanh.
UniformRandomNumber	<ul style="list-style-type: none"> • One IWork vector, RandSeed. • One RWork vector, NextOutput. • Seed ParamSetting. • Minimum parameter. • Maximum parameter.

Table A-21: Model.rtw Block Specific Records (Continued)

Block Type	Properties
UnitDelay	<ul style="list-style-type: none">• One DWork vector, PreviousInput, used to produce the output.• X0 parameter - the initial condition.
VariableTransportDelay	<ul style="list-style-type: none">• InitialInput ParamSetting.• BufferSize ParamSetting.• DiscreteInput ParamSetting.• One IWork vector, BufferIndices.• One PWork vector, TUBuffer.• DelayTime parameter.
Width	No block specific records.
ZeroPole	See Model.rtw Linear Block Specific Records.
ZeroOrderHold	No block specific records.

Linear Block Specific Records

This table describes the block specific records written for the Simulink linear blocks. These fields are common to all the discrete and continuous state space, transfer function, and discrete filter blocks previously discussed.

Table A-22: Model.rtw Linear Block Specific Records

Parameter {	Vector of nonzero terms of the A matrix if realization is sparse, otherwise it is the first row of the A matrix.
Name	"Amatrix"
Value	Vector that could be of zero length.
String	" "
StringType	"Computed"
}	
Parameter {	Vector of nonzero terms of the B matrix.
Name	"Bmatrix"
Value	Vector that could be of zero length.
String	" "
StringType	"Computed"
}	
Parameter {	Vector of nonzero terms of the C matrix if realization is sparse, else it is the full C 2-D matrix.
Name	"Cmatrix"
Value	Vector that could be of zero length.
String	" "
StringType	"Computed"
}	
Parameter {	Vector of nonzero terms of the D matrix.
Name	"Dmatrix"
Value	Vector that could be of zero length.
String	""
StringType	"Computed"
}	
Parameter {	Initial condition vector or [].
Name	"X0"

Table A-22: Model.rtw Linear Block Specific Records (Continued)

Val ue	Vector that could be of zero length.
String	""
StringType	"Computed"
}	
ParamSettings {	
NumNonZeroAInRow	Vector of the number of nonzero elements in each row of the A matrix.
ColIdxOfNonZeroA	Column index of the nonzero elements in the A matrix.
NumNonZeroBInRow	Vector of the number of nonzero elements in each row of the B matrix.
ColIdxOfNonZeroB	Column index of the nonzero elements in the B matrix.
NumNonZeroCInRow	Vector of the number of nonzero elements in each row of the C matrix.
ColIdxOfNonZeroC	Column index of the nonzero elements in the C matrix.
NumNonZeroDInRow	Vector of the number of nonzero elements in each row of the D matrix.
ColIdxOfNonZeroD	Column index of the nonzero elements in the D matrix.
}	

TLC Error Handling

Generating Errors from TLC-Files	B-2
Usage Errors	B-2
Fatal (Internal) TLC Coding Errors	B-2
Formatting Error Messages	B-3
TLC Error Messages	B-5
TLC Function Library Error Messages	B-32

Generating Errors from TLC-Files

To generate errors from TLC files, use the `%exit` directive. The two types of errors are:

Usage errors	These can be caused by incorrect models.
Internal coding errors	These <i>cannot</i> be caused by incorrect models.

Usage Errors

Usage errors are errors that can happen by incorrect models or other 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 directory `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

```
Li bBl ockReportFatal Error(block, "fatal coding error message")
```

where *block* is the offending block record (or [] if the block is already scoped), and

```
Li bReportFatal Error("fatal coding error message")
```

for error messages that are not block specific. For example, to add assert code you could use

```
%i f TYPE(argument) != "Number"
    %<Li bBl ockReportFatal Error(block, "unexpected argument type")
%endi f
```

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 error 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.

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

```
%setcommandswit ch "-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 (t1 c. 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 5-32 for more information.

A %switch statement can only have one %default

The user has written a `%switch` statement with multiple `%default` cases, as in the following example.

```
%switch expr
  %case 1
    code...
    %break
  %default

more code...
  %break
  %default %% error
    even more code...
    %break
%endswitch
```

A language choice must be made using the %language directive prior to using GENERATE or GENERATE_TYPE

To use the `GENERATE` or `GENERATE_TYPE` built-in functions, the Target Language Compiler requires that you first specify the language being generated. 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.

```
%i f expr
  code. . .
%el se
  more code. . .
%el se      %% error
  even mode code. . .
%endi f
```

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_FI LENAME
EXI STS	GENERATE_FUNCTI ON_EXI STS
FEVAL	GENERATE_TYPE
FI LE_EXI STS	GET_COMMAND _SWI TCH
FORMAT	I DNUM
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:

- I SALI AS
- REMOVEFI ELD
- FI ELDNAMES
- I SFI ELD
- GETFI ELD
- SETFI ELD.

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 -O 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. e.g.

```
%<foo(%<expr>) >
```

Instead, do the following:

```
%assi gn 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,

```
%<Compi l edModel . Syst em[ Sysi dx] . Bl ock[ Bl kI dx] . Name +
"Hel l o">
```

is illegal, whereas

```
%<Compi l edModel . Syst em[ Sysi dx] . Bl ock[ Bl kI dx] . Name + \
"Hel l o">
```

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 %real format 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 *op*.

Unary operators - and + require numeric types. Unary operator ~ requires an integral type. Unary operator ! requires a Boolean type.

Invalid type for unary *operator*

This error occurs for the following operators under the given conditions.

Operator	Reason for Error
!	Operand to the logical not operator (!) must be a Number, Real, or Boolean.
-	Operand to the unary negation operator (-) must be a Number or Real.
+	Operand to the unary plus operator (+) must be a Number or Real.
~	Operand to the bitwise negation operator (~) must be a Number.

Invalid type *type*

An invalid type was passed to a builtin function.

It is illegal to return a function from a function

A function value cannot be returned from a function call.

License checkout failed. In order to generate Ada, you must purchase a license. Please contact your MathWorks sales representative.

This error occurs when you do not have a valid license available for generating Ada code.

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 Chapter 5, “Directives and Built-in Functions,” for information on the syntax.

Syntax error detected in EXISTS function called with "string"

The EXISTS function parses and evaluates the string passed to it. The function reports this error when it is unable to parse the input string successfully. To better diagnose the error, you can try to define the symbol and then type the identical expression inside an expansion directive. For example,

```
%i f EXISTS( "x[100].y" )  
%% If this fails, try  
%<x[100].y>  
%% In order to receive a better diagnosis of the problem.
```

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 %i f must be a numeric type. For example,

```
%i f [ 1 2 3 ]
```

```
...
```

```
%endi f
```

%i f cannot accept a vector as input.

The %implements directive expects a string or string vector as the list of languages

You can use the %i m p l e m e n t s 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 %i m p l e m e n t s directive.

The %implements directive specifies *type* as the type where *type* was expected

The type specified in the %i m p l e m e n t s 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 %i m p l e m e n t s * directive, as in

```
%i m p l e m e n t s * "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 %i m p l e m e n t s directive must exactly match the %l a n g u a g e 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
<code>%if</code>	<code>%endif</code>	Conditional inclusion
<code>%for</code>	<code>%endfor</code>	Looping
<code>%foreach</code>	<code>%endforeach</code>	Looping
<code>%roll</code>	<code>%endroll</code>	Loop Rolling
<code>%with</code>	<code>%endwith</code>	Scoping directive
<code>%switch</code>	<code>%endswitch</code>	Switch directive
<code>%function</code>	<code>%endfunction</code>	Function declaration directive
<code>{</code>	<code>}</code>	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 5-46.

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 valued argument

These functions expect a Real value as the input argument.

The language being implemented cannot be changed within a block template file

You cannot change the language using the **%l** 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 **%l** 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 to 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

%case	%generatefile
%closefile	%if
%continue	%implements
%copyrecord	%include
%createrecord	%language
%default	%matlab
%define	%mergerecord
%else	%openfile
%elseif	%realformat
%endbody	%return
%endfor	%roll
%endforeach	%selectfile
%endfunction	%setcommandswitch
%endif	%switch
%endroll	%trace
%endswitch	%undef
%endwith	%warning
%error	%with
%exit	

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 multi-line comment.

A multi-line (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.
-d[a c n o]	<p>Invoke the TLC's debug mode.</p> <p>- da will make TLC execute any %assert directives.</p> <p>- dc will invoke TLC's command line debugger.</p> <p>- dn will cause TLC to produce log files indicating which lines were and were not hit during compilation.</p> <p>- do will disable TLC's debugging behavior.</p>
-a<i dent>=<expressi on>	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 RTW to set options like inlining of parameters, file size limits, etc.

Message	Description
- p<number>	Print a '.' indicating progress for every <number> of TLC primitive operations executed.
- l i n t	Perform some simple performance checks and collect some runtime statistics.
- x0	Parse a TLC file, but not execute it.

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

 %<Functi on>

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-documenting 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	C-2
Getting Started	C-2
Creating a TAGS File	C-2

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`.

TLC code is much more readable using the color-coded syntax feature in Emacs.

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 directory where your `.t1c` files are located. The `etags` command is located the `emacs_root/bin` directory.

Symbols

- ! 5-21
- 5-21
- 5-22
- != 5-23
- % 5-2, 5-19, 5-22
- & 5-23
- && 5-23
- () 5-21
- * 5-21
- + 5-21, 5-22
- , 5-24
- . 5-21
- ... 5-16
- .c file 1-5
- .h file 1-5
- .log 6-11
- .rtw file 1-5
- .rtw file structure 4-2
- / 5-22
- :: 5-21, 5-52
- < 5-23
- << 5-22
- <= 5-23
- == 5-23
- > 5-23
- >= 5-23
- >> 5-22
- ? : 5-24
- \ 5-16
- ^ 5-23
- _prm.h file 1-5
- _reg.h file 1-5
- | 5-23
- || 5-23
- ~ 5-21

A

- %addincludepath 5-35
- array index 5-21
- %assert 5-36
- assert
 - adding B-2
- %assign 5-51, 7-26
 - defining parameters 3-18

B

- block
 - customizing Simulink 5-32
- block function 7-31
 - InitializeConditions 7-37
 - Start 7-37
- block target file 1-4, 3-15, 3-17, 3-22, 7-31
 - function in 7-27
 - mapping 3-17
 - writing 7-32
- BlockInstanceSetup 7-32
- block-scoped variable 5-55
- BlockTypeSetup 7-33
- %body 5-28
- Boolean 5-17
- %break 5-27, 5-28
- %continue 5-27
- buffer
 - close 5-35
 - writing 5-34
- built-in functions 5-37
 - CAST 5-38
 - EXISTS 5-38
 - FEVAL 5-39
 - FIELDNAMES 5-39
 - FILE_EXISTS 5-39

FORMAT 5-39
GENERATE 5-40
GENERATE_FILENAME 5-40
GENERATE_FORMATTED_VALUE 5-40
GENERATE_FUNCTION_EXISTS 5-40
GENERATE_TYPE 5-41
GENERATE_TYPE_FUNCTION_EXISTS 5-41
GET_COMMAND_SWITCH 5-41
GETFIELD 5-40
IDNUM 5-41
IMAG 5-41
INT16MAX 5-41
INT16MIN 5-41
INT32MAX 5-41
INT32MIN 5-41
INT8MAX 5-41
INT8MIN 5-41
ISALIAS 5-42
ISEMPTY 5-42
ISEQUAL 5-42
ISFIELD 5-42
ISFINITE 5-42
ISINF 5-42
ISNAN 5-42
NULL_FILE 5-43
NUMTLCFILES 5-43
OUTPUT_LINES 5-43
REAL 5-43
REMOVEFIELD 5-43
ROLL_ITERATIONS 5-43
SETFIELD 5-43
SIZE 5-44
STDOUT 5-44
STRING 5-44
STRINGOF 5-44
SYSNAME 5-45
TLC_FALSE 5-45

TLC_TIME 5-45
TLC_TRUE 5-45
TLC_VERSION 5-46
TLCFILES 5-45
TYPE 5-46
UINT16MAX 5-46
UINT32MAX 5-46
UINT8MAX 5-46
WHITESPACE 5-46
WILL_ROLL 5-46

C

C MEX S-function 1-4
%case 5-27
CAST 5-38
%closefile 5-34
code
 intermediate 3-17
code coverage 6-11
code generation 1-9
coding conventions 7-26
comment
 target language 5-15
CompiledModel 4-3
Compiler
 Target Language (TLC) 1-2
Complex 5-17
Complex32 5-17
conditional
 inclusion 5-25
 operator 5-20
constant
 integer 5-19
 string 5-19
continuation
 line 5-16

%continue 5-28
 customizing
 code generation 3-17
 Simulink block 5-32

D

debug
 message 5-36
 debugger 6-2, 6-6
 example session 6-9
 using 6-3
 debugger commands
 viewing 6-8
 debugging tips 6-2
 %default 5-27
 Derivatives 7-39
 directive 3-18, 5-2
 object-oriented 5-32
 splitting 5-16
 directives
 %% 5-2
 %<expr> 5-3
 %addincludepath 5-8
 %addtorecord 5-6
 %assert 5-4
 %assign 5-5
 %break 5-4
 %case 5-4
 %closefile 5-15
 %copyrecord 5-7
 %createrecord 5-6
 %default 5-4
 %else 5-4
 %elseif 5-4
 %endforeach 5-13
 %endfunction 5-12

%endif 5-4
 %endroll 5-9
 %endswitch 5-4
 %endwith 5-4
 %error 5-5
 %exit 5-5
 %filescope 5-8
 %for 5-14
 %foreach 5-13
 %function 5-12
 %generatefile 5-8
 %if 5-4
 %implements 5-8
 %include 5-8
 %language 5-7
 %matlab 5-2
 %mergerecord 5-7
 %openfile 5-15
 %realformat 5-7
 %return 5-12
 %roll 5-9
 %selectfile 5-15
 %setcommandswitch 5-4
 %switch 5-4
 %trace 5-5
 %warning 5-5
 %with 5-4
 /% text %/ 5-2
 Diabole 7-36
 dynamic scoping 5-56

E

%else 5-26
 %elseif 5-26
 Enable 7-35
 %endbody 5-28

- `%endfor` 5-28
- `%endforeach` 5-27
- `%endfunction` 5-59
- `%endif` 5-26
- `%endswitch` 5-27
- `%endwith` 5-55
- `%error` 5-36
- error
 - formatting messages B-3
 - internal B-2
 - usage B-2
- error message 5-36
 - Target Language Compiler B-5
- EXISTS 5-38
- `%exit` 5-36
- expressions 5-19
 - operators in 5-19
 - precedence 5-19

F

- FEVAL 5-39
- FIELDNAMES 5-39
- File 5-17
- file
 - `.c` 1-5
 - `.h` 1-5
 - `.rtw` 1-5
 - `_prmh` 1-5
 - `_reg.h` 1-5
 - appending 5-35
 - block target 1-4, 3-17, 3-22
 - close 5-35
 - inline 5-35
 - model description. *See* `model.rtw`
 - model-wide target 3-18
 - system target 3-16

- target 1-4, 3-17
- target language 3-17
 - used to customize code 3-17
 - writing 5-34
- FILE_EXISTS 5-39
- `%for` 5-28
- `%foreach` 5-27
- FORMAT 5-39
- formatting 5-25
- frame signal A-6
- Function 5-17
- `%function` 5-59
- function
 - C MEX S-function 1-4
 - call 5-21
 - GENERATE 5-33
 - GENERATE_TYPE 5-33
 - library 7-29
 - output 5-60
 - target language 5-59
 - Target Language Compiler 5-37–5-46
- functions
 - obsolete 9-3

G

- Gaussian 5-17
- Gaussian, Unsigned 5-19
- GENERATE 5-33, 5-40
- GENERATE_FILENAME 5-40
- GENERATE_FORMATTED_VALUE 5-40
- GENERATE_FUNCTION_EXISTS 5-40
- GENERATE_TYPE 5-33, 5-41
- GENERATE_TYPE_FUNCTION_EXISTS 5-41
- `%generatefile` 5-32
- GET_COMMAND_SWITCH 5-41
- GETFIELD 5-40

- I**
- identifier 7-26
 - changing 5-51
 - defining 5-51
- IDNUM 5-41
- %i f %endi f 5-26
- IMAG 5-41
- %i mplements 5-32
- %i nclude 5-35
- inclusion
 - conditional 5-25
 - multiple 5-27
- index 5-21
- Initialize 7-37
- InitializeConditions 7-37
- inlining S-function 7-4
 - advantages 1-14
- input file control 5-35
- INT16MAX 5-41
- INT16MIN 5-41
- INT32MAX 5-41
- INT32MIN 5-41
- INT8MAX 5-41
- INT8MIN 5-41
- integer constant 5-19
- intermediate code 3-17
- ISALIAS 5-42
- ISEMPTY 5-42
- ISEQUAL 5-42
- ISFIELD 5-42
- ISFINITE 5-42
- SINF 5-42
- ISNAN 5-42
- L**
- %l anguage 5-32
- lc, definition 9-5
- library functions
 - LibAddToModel Sources 9-27
 - LibBlockContinuousState 9-22
 - LibBlockDiscreteState 9-23
 - LibBlockDWork 9-22
 - LibBlockDWorkAddr 9-22
 - LibBlockDWorkDataTypeId 9-22
 - LibBlockDWorkDataTypeName 9-23
 - LibBlockDWorkIsComplex 9-23
 - LibBlockDWorkName 9-23
 - LibBlockDWorkUsedAsDiscreteState 9-23
 - LibBlockDWorkWidth 9-23
 - LibBlockInputSignal 9-10
 - LibBlockInputSignalAddr 9-12
 - LibBlockInputSignalBufferDstPort 9-39
 - LibBlockInputSignalDataTypeId 9-13
 - LibBlockInputSignalDataTypeName 9-13
 - LibBlockInputSignalDimensions 9-14
 - LibBlockInputSignalIsComplex 9-14
 - LibBlockInputSignalIsFrameData 9-14
 - LibBlockInputSignalNumDimensions 9-14
 - LibBlockInputSignalStorageClass 9-40
 - LibBlockInputSignalStorageTypeQualifier
 - 9-40
 - LibBlockInputSignalWidth 9-14
 - LibBlockWork 9-24
 - LibBlockMatrixParameter 9-18
 - LibBlockMatrixParameterAddr 9-18
 - LibBlockMode 9-24
 - LibBlockOutputSignal 9-15
 - LibBlockOutputSignalAddr 9-15
 - LibBlockOutputSignalDataTypeId 9-16
 - LibBlockOutputSignalDataTypeName 9-16
 - LibBlockOutputSignalDimensions 9-17
 - LibBlockOutputSignalIsComplex 9-17
 - LibBlockOutputSignalIsFrameData 9-17

- Li bBl ockOutputSi gnal IsGl obal 9-40
- Li bBl ockOutputSi gnal IsInBl ockIO 9-41
- Li bBl ockOutputSi gnal IsVal i dLVal ue 9-41
- Li bBl ockOutputSi gnal NumDi mensi ons 9-17
- Li bBl ockOutputSi gnal StorageCl ass 9-41
- Li bBl ockOutputSi gnal StorageTypeQual i fi e
r 9-41
- Li bBl ockOutputSi gnal Wi dth 9-17
- Li bBl ockParameter 9-19
- Li bBl ockParameterAddr 9-20
- Li bBl ockParameterDataTypeId 9-21
- Li bBl ockParameterDataTypeName 9-21
- Li bBl ockParameterIsCompl ex 9-21
- Li bBl ockParameterSi ze 9-21
- Li bBl ockPWork 9-24
- Li bBl ockReportError 9-25
- Li bBl ockReportFatal Error 9-25
- Li bBl ockReportWarni ng 9-25
- Li bBl ockRWork 9-24
- Li bBl ockSrcSi gnal Bl ock 9-42
- Li bBl ockSrcSi gnal IsDi screte 9-42
- Li bBl ockSrcSi gnal IsGl obal AndModi fi abl e
9-43
- Li bBl ockSrcSi gnal IsInvari ant 9-43
- Li bCacheDefi ne 9-27
- Li bCacheExtern 9-28
- Li bCacheFuncti onPrototype 9-28
- Li bCacheIncl udes 9-28
- Li bCacheTypedefs 9-29
- Li bCal l FCSS 9-36
- Li bGetBl ockPath 9-26
- Li bGetDataTypeCompl exNameFromId 9-37
- Li bGetDataTypeEnumFromId 9-37
- Li bGetDataTypeNameFromId 9-37
- Li bGetFormattedBl ockPath 9-26
- Li bGetGl obal TIDFromLocal SFcnTID 9-30
- Li bGetNumSFcnSampl eTi mes 9-31

- Li bGetSFcnTIDType 9-32
- Li bGetT 9-37
- Li bGetTaskTi meFromTID 9-32
- Li bIsCompl ex 9-37
- Li bIsCont i nuous 9-32
- Li bIsDi screte 9-32
- Li bIsFi rstIni tCond 9-38
- Li bIsSFcnSampl eHi t 9-33
- Li bIsSFcnSi ngl eRate 9-34
- Li bIsSFcnSpeci al Sampl eHi t 9-34
- Li bMaxIntVal ue 9-38
- Li bMi nIntVal ue 9-38

M

- macro

- expansion 5-21

- makefile

- template 1-4

- Matrix 5-18

- mdl Deri vati ves (S-function) 7-4

- mdl Ini ti al i zeCondi ti ons 7-4

- mdl Ini ti al i zeSampl eTi mes 7-4

- mdl Ini ti al i zeSi zes 7-4

- mdl Out puts (S-function) 7-4

- Mdl Start

- Ini ti al i zeCondi ti ons 7-37

- Mdl Termi nate

- Termi nate 7-40

- mdl Termi nate (S-function) 7-4

- mdl Update (S-function) 7-4

- model description file. *See* model . rtw

- model . rtw

- changes A-6

- model . rtw file 1-3, 5-37

- parameter-value pair 4-2

- record 4-2

- scope 4-3
- structure 4-2
- model-wide target file 3-18
- modifier
 - Output 5-60
 - void 5-60
- multiple inclusion 5-27

N

- negation operator 5-21
- nested function
 - scope within 5-62
- NULL_FILE 5-43
- Number 5-18
- NUMTLCFILES 5-43

O

- object-oriented directive 5-32
- obsolete functions 9-3
- %openfile 5-34
- operations
 - precedence 5-21
- operator 5-19
 - :: 7-27
 - conditional 5-20
 - negation 5-21
- output file control 5-34
- Output modifier 5-60
- OUTPUT_LINES 5-43
- Outputs 7-38

P

- parameter
 - defining 3-18

- value pair 4-2
- paramIdx 9-7
- path
 - specifying absolute 5-35
 - specifying relative 5-36
- portIdx, definition 9-5
- precedence
 - expressions 5-19
 - operations 5-21
- profiler 6-14
 - using 6-14
- program 3-17

R

- Range 5-18
- REAL 5-43
- Real 5-18
- Real 32 5-18
- %real format 5-25
- Real-Time Workshop 1-2
- record 3-9, 4-2
- REMOVEFIELD 5-43
- resolving variables 5-56
- %return 5-59, 5-64
- %roll 5-29
- ROLL_ITERATIONS 5-43
- rt 7-28
- rt_ 7-28
- RTW
 - identifier 7-26

S

- Scope 5-18
- scope 5-55
 - accessing values in 4-3

- closing 5-64
- dynamic 5-56
- function 5-21
- model . rtw file 4-3
- within function 5-60
- search path 5-66
 - adding to 5-35
 - overriding 5-66
 - sequence 5-36
 - specifying absolute 5-35
 - specifying relative 5-36
- %selectfile 5-34
- SETFIELD 5-43
- S-function
 - advantage of inlining 1-14
 - C MEX 1-4
 - inlining 7-4
 - user-defined 7-40
- S-function record A-71
- sigIdx 9-6
- Simulink
 - and Real-Time Workshop 1-2
 - generating code 1-5
- SIZE 5-44
- Special 5-18
- Start 7-36
- stateIdx 9-7
- STDOUT 5-44
- STRING 5-44
- String 5-18
- string constant 5-19
- STRINGOF 5-44
- substitution
 - textual 5-19
- Subsystem 5-18
- %switch 5-26
- syntax 5-2

- SYS_NAME 5-45
- system target file 3-16

T

- target file 1-4, 3-14, 3-17
 - and customizing code 3-17
 - block 3-17, 3-22, 7-31
 - model-wide 3-18
 - naming 5-66
 - system 3-16
- target language 3-8
 - comment 5-15
 - directive 3-18, 5-2
 - expression 5-19–5-24
 - file 5-2
 - formatting 5-25
 - function 5-59
 - line continuation 5-16
 - program 3-17
 - syntax 5-2
 - value 5-17–5-19
- Target Language Compiler
 - command line arguments 5-65
 - directives 5-2–5-15
 - error messages B-5
 - function library 7-29
 - introducing 1-2
 - switches 5-65
 - uses of 1-7
 - variables 7-28
- template makefile 1-4
- Terminate 7-40
- textual substitution 5-19
- TLC code
 - debugging tips 6-2
- TLC coverage option 6-11

TLC debugger 6-2
TLC debugger commands 6-6
TLC profiler 6-14
TLC program 3-17
TLC_FALSE 5-45
TLC_TIME 5-45
TLC_TRUE 5-45
TLC_VERSION 5-46
TLCFILES 5-45
%trace 5-36
tracing 5-36
TYPE 5-46

U

ucv, definition 9-5
UINT16MAX 5-46
UINT32MAX 5-46
UINT8MAX 5-46
Unsigned 5-18
Unsigned Gaussian 5-19
Update 7-39

V

values 5-17
variables
 block-scoped 5-55
 global 7-27
 local 7-27
Vector 5-19
void modifier 5-60

W

%warning 5-36
warning message 5-36

WHITE_SPACE 5-46
WILL_ROLL 5-46
%with 5-55

Z

zero-crossing
 reset code 7-39