Target Language Compiler™

For Use with Real-Time Workshop®

Modeling

Simulation

Implementation

The MathWorks

Reference Guide

Version 5

How to Contact The MathWorks:

	`www.mathworks.com`	Web
	`comp.soft-sys.matlab`	Newsgroup
@	`support@mathworks.com`	Technical support
	`suggest@mathworks.com`	Product enhancement suggestions
	`bugs@mathworks.com`	Bug reports
	`doc@mathworks.com`	Documentation error reports
	`service@mathworks.com`	Order status, license renewals, passcodes
	`info@mathworks.com`	Sales, pricing, and general information
	508-647-7000	Phone
	508-647-7001	Fax
	The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail

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

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

Contents

1 Introducing the Target Language Compiler

Introduction **1-2**
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**

Advantages of Inlining S-Functions **1-13**
Motivations **1-13**
Inlining Process **1-14**
Search Algorithm for Locating Target Files **1-15**
Availability for Inlining and Noninlining **1-15**

New Features and Compatibility Issues in Versions 4.0, 4.1, and 5.0 . . . . **1-16**
New Features Added in Version 5.0 . . . . **1-16**
New Features Added in Version 4.1 . . . . **1-17**
New Features Added in Version 4.0 . . . . **1-18**
Compatibility Issues **1-19**

Where to Go from Here **1-23**
Related Manuals **1-23**

2 Getting Started

Code Architecture **2-2**

model.rtw and Target Language Compiler Overview **2-4**
The Target Language Compiler Process **2-4**

Inlining S-Function Concepts **2-6**
Noninlined S-Function **2-6**
Types of Inlining **2-7**
Fully Inlined S-Function Example **2-8**
Wrapper Inlined S-Function Example **2-10**

3 A TLC Tutorial

Introduction **3-2**

Reading Record Files with TLC **3-3**
Structure of Record Files **3-3**
Interpreting Records **3-4**
Anatomy of a TLC Script **3-6**
Modify read-guide.tlc **3-13**

Processing model.rtw Files with TLC Scripts **3-18**
Getting Started **3-18**
Counting Blocks and Subsystems **3-19**
Listing Block Names **3-21**
Passing and Using a Parameter **3-22**

Inlining S-Functions with TLC **3-25**
Noninlined Code Generation **3-25**
Why Use TLC to Implement S-functions? **3-27**
Creating an Inlined S-Function **3-28**
Scalar or Vector, It’s the Same **3-31**

Exploring Variable Names and Loop Rolling **3-32**
Getting Started **3-32**
Modify the Model **3-33**
Change the Loop Rolling Threshold **3-35**
More on TLC Loop Rolling **3-37**

Debugging Your TLC Code **3-40**
Getting Started **3-40**
Generate and Run Code from the Model **3-42**
Start the Debugger and Explore Commands **3-44**
Debug gain.tlc **3-45**
Fix the Bug and Verify the Fix **3-47**

Using TLC Code Coverage to Aid Debugging **3-49**
Getting Started **3-49**
Open the Model and Generate Code **3-50**
Log Without Inline Parameters **3-51**

Wrapping User Code with TLC **3-53**
Why Wrap? **3-53**
Getting Started **3-55**
Generate Code Without a Wrapper **3-56**
Generate Code Using a Wrapper **3-59**

4 Code Generation Architecture

Build Process **4-2**
A Basic Example **4-2**

Invoking Code Generation **4-9**
The rtwgen Command **4-9**
The tlc Command **4-9**

Configuring TLC **4-11**

Code Generation Concepts **4-13**
Output Streams **4-13**
Variable Types **4-14**
Records **4-14**
Record Aliases **4-16**

TLC Files **4-19**
Available Target Files **4-19**
Summary of Target File Usage **4-23**
System Target Files **4-24**
Block Target Files **4-25**
Block Target File Mapping **4-26**

Data Handling with TLC: An Example **4-27**
Matrix Parameters in Real-Time Workshop **4-27**

5 Contents of model.rtw

Model.rtw File Overview **5-2**
Using Scopes in the model.rtw File **5-3**
Object Information in the model.rtw File **5-6**

Using Library Functions to Access model.rtw Contents . . **5-10**
Caution Against Directly Accessing Record Fields **5-10**
Exception to Using the Library Functions **5-11**

6 Directives and Built-in Functions

Compiler Directives **6-2**
Syntax **6-2**
Comments **6-17**
Line Continuation **6-18**

Target Language Values **6-19**
Target Language Expressions **6-21**
Formatting **6-27**
Conditional Inclusion **6-28**
Multiple Inclusion **6-29**
Object-Oriented Facility for Generating Target Code **6-34**
Output File Control **6-36**
Input File Control **6-37**
Asserts, Errors, Warnings, and Debug Messages **6-38**
Built-In Functions and Values **6-39**
TLC Reserved Constants **6-50**
Identifier Definition **6-51**
Variable Scoping **6-55**
Target Language Functions **6-65**

Command Line Arguments **6-70**
Filenames and Search Paths **6-72**

7 **Debugging TLC Files**

About the TLC Debugger **7-2**
Tips for Debugging TLC Code **7-2**

Using the TLC Debugger **7-3**
Tutorial **7-3**
TLC Debugger Command Summary **7-6**

TLC Coverage **7-10**
Using the TLC Coverage Option **7-10**

TLC Profiler **7-13**
Using the Profiler **7-13**

8 Inlining S-Functions

Writing Block Target Files to Inline S-Functions **8-2**
Fully Inlined S-Functions **8-2**
Function-Based or Wrappered Code Generation **8-2**

Inlining C MEX S-Functions **8-4**
S-Function Parameters **8-5**
A Complete Example **8-6**

Inlining M-File S-Functions **8-17**

Inlining Fortran (F-MEX) S-Functions **8-19**

TLC Coding Conventions **8-23**

Block Target File Methods **8-28**
Block Target File Mapping **8-28**
Block Functions **8-28**

Loop Rolling **8-36**

Error Reporting **8-39**

9 TLC Function Library Reference

Obsolete Functions **9-2**

Target Language Compiler Functions **9-4**
Common Function Arguments **9-4**

Input Signal Functions **9-9**
LibBlockInputSignal(portIdx, ucv, lcv, sigIdx) **9-9**
Input Arguments **9-10**
General Usage **9-11**

LibBlockInputSignalAddr(portIdx, ucv, lcv, sigIdx) **9-16**
LibBlockInputSignalConnected(portIdx) **9-16**
LibBlockInputSignalDataTypeId(portIdx) **9-17**
LibBlockInputSignalDataTypeName(portIdx, reim) **9-17**
LibBlockInputSignalDimensions(portIdx) **9-17**
LibBlockInputSignalIsComplex(portIdx) **9-17**
LibBlockInputSignalIsFrameData(portIdx) **9-17**
LibBlockInputSignalLocalSampleTimeIndex(portIdx) **9-18**
LibBlockInputSignalNumDimensions(portIdx) **9-18**
LibBlockInputSignalOffsetTime(portIdx) **9-18**
LibBlockInputSignalSampleTime(portIdx) **9-18**
LibBlockInputSignalSampleTimeIndex(portIdx) **9-18**
LibBlockInputSignalWidth(portIdx) **9-18**

Output Signal Functions **9-19**
LibBlockOutputSignal(portIdx, ucv, lcv, sigIdx) **9-19**
LibBlockOutputSignalAddr(portIdx, ucv, lcv, sigIdx) **9-19**
LibBlockOutputSignalBeingMerged(portIdx) **9-20**
LibBlockOutputSignalConnected(portIdx) **9-20**
LibBlockOutputSignalDataTypeId(portIdx) **9-20**
LibBlockOutputSignalDataTypeName(portIdx, reim) **9-20**
LibBlockOutputSignalDimensions(portIdx) **9-21**
LibBlockOutputSignalIsComplex(portIdx) **9-21**
LibBlockOutputSignalIsFrameData(portIdx) **9-21**
LibBlockOutputSignalLocalSampleTimeIndex(portIdx) **9-21**
LibBlockOutputSignalNumDimensions(portIdx) **9-21**
LibBlockOutputSignalOffsetTime(portIdx) **9-21**
LibBlockOutputSignalSampleTime(portIdx) **9-22**
LibBlockOutputSignalSampleTimeIndex(portIdx) **9-22**
LibBlockOutputSignalWidth(portIdx) **9-22**

Parameter Functions **9-23**
LibBlockMatrixParameter
(param,rucv,rlcv,ridx,cucv,clcv,cidx) **9-23**
LibBlockMatrixParameterAddr
(param,rucv,rlcv,ridx,cucv,clcv,cidx) **9-23**
LibBlockMatrixParameterBaseAddr(param) **9-23**
LibBlockParameter(param, ucv, lcv, sigIdx) **9-24**
LibBlockParameterAddr(param, ucv, lcv, idx) **9-25**
LibBlockParameterBaseAddr(param) **9-26**

LibBlockParameterDataTypeId(param) **9-26**
LibBlockParameterDataTypeName(param, reim) **9-26**
LibBlockParameterDimensions(param) **9-26**
LibBlockParameterIsComplex(param) **9-27**
LibBlockParameterSize(param) **9-27**

Block State and Work Vector Functions **9-28**
LibBlockContinuousState(ucv, lcv, idx) **9-28**
LibBlockDWork(dwork, ucv, lcv, sigIdx) **9-28**
LibBlockDWorkAddr(dwork, ucv, lcv, idx) **9-28**
LibBlockDWorkDataTypeId(dwork) **9-28**
LibBlockDWorkDataTypeName(dwork, reim) **9-28**
LibBlockDWorkIsComplex(dwork) **9-29**
LibBlockDWorkName(dwork) **9-29**
LibBlockDWorkStorageClass(dwork) **9-29**
LibBlockDWorkUsedAsDiscreteState(dwork) **9-29**
LibBlockDWorkWidth(dwork) **9-29**
LibBlockDiscreteState(ucv, lcv, idx) **9-29**
LibBlockIWork(definediwork, ucv, lcv, idx) **9-29**
LibBlockMode(ucv, lcv, idx) **9-30**
LibBlockNonSampledZC(ucv, lcv, NonSampledZCIdx) **9-30**
LibBlockPWork(definedpwork, ucv, lcv, idx) **9-30**
LibBlockRWork(definedrwork, ucv, lcv, idx) **9-30**

Block Path and Error Reporting Functions **9-31**
LibBlockReportError(block,errorstring) **9-31**
LibBlockReportFatalError(block,errorstring) **9-31**
LibBlockReportWarning(block,warnstring) **9-31**
LibGetBlockPath(block) **9-32**
LibGetFormattedBlockPath(block) **9-32**

Code Configuration Functions **9-33**
LibAddToCommonIncludes(incFileName) **9-33**
LibAddToModelSources(newFile) **9-33**
LibCacheDefine(buffer) **9-33**
LibCacheExtern(buffer) **9-34**
LibCacheFunctionPrototype(buffer) **9-34**
LibCacheIncludes(buffer) **9-35**
LibCacheTypedefs(buffer) **9-35**
LibRegisterGNUMathFcnPrototypes() **9-35**

LibRegisterISOCMathFcnPrototypes() **9-36**
LibRegisterMathFcnPrototype(RTWName, RTWType, IsExprOK, IsCplx, NumInputs, FcnName, FcnType, HdrFile) **9-36**

Sample Time Functions **9-37**
LibGetGlobalTIDFromLocalSFcnTID(sfcnTID) **9-37**
LibGetNumSFcnSampleTimes(block) **9-39**
LibGetSFcnTIDType(sfcnTID) **9-39**
LibGetTaskTimeFromTID(block) **9-39**
LibIsContinuous(TID) **9-39**
LibIsDiscrete(TID) **9-40**
LibIsSFcnSampleHit(sfcnTID) **9-40**
LibIsSFcnSingleRate(block) **9-41**
LibIsSFcnSpecialSampleHit(sfcnSTI, sfcnTID) **9-41**
LibPortBasedSampleTimeBlockIsTriggered(block) **9-42**
LibSetVarNextHitTime(block,tNext) **9-42**

Other Useful Functions **9-43**
LibCallFCSS(system, simObject, portEl, tidVal) **9-43**
LibGenConstVectWithInit(data, typeId, varId) **9-44**
LibGetDataTypeComplexNameFromId(id) **9-44**
LibGetDataTypeEnumFromId(id) **9-44**
LibGetDataTypeIdAliasedToFromId(id) **9-44**
LibGetDataTypeIdResolvesToFromId(id) **9-44**
LibGetDataTypeNameFromId(id) **9-45**
LibGetT() **9-45**
LibIsMajorTimeStep() **9-45**
LibIsMinorTimeStep() **9-45**
LibIsComplex(arg) **9-45**
LibIsFirstInitCond(s) **9-46**
LibMaxIntValue(dtype) **9-46**
LibMinIntValue(dtype) **9-46**

Advanced Functions **9-47**
LibBlockInputSignalBufferDstPort(portIdx) **9-47**
LibBlockInputSignalStorageClass(portIdx, idx) **9-48**
LibBlockInputSignalStorageTypeQualifier(portIdx, idx) **9-48**
LibBlockOutputSignalIsGlobal(portIdx) **9-48**
LibBlockOutputSignalIsInBlockIO(portIdx) **9-49**

LibBlockOutputSignalIsValidLValue(portIdx) **9-49**
LibBlockOutputSignalStorageClass(portIdx) **9-49**
LibBlockOutputSignalStorageTypeQualifier(portIdx) **9-49**
LibBlockSrcSignalBlock(portIdx, idx) **9-49**
LibBlockSrcSignalIsDiscrete(portIdx, idx) **9-50**
LibBlockSrcSignalIsGlobalAndModifiable(portIdx, idx) **9-50**
LibBlockSrcSignalIsInvariant(portIdx, idx) **9-51**
LibCreateHomogMathFcnRec(FcnName, FcnTypeId) **9-51**
LibGetMathConstant(ConstName,ioTypeId) **9-51**
LibMathFcnExists(RTWFcnName, RTWFcnTypeId) **9-51**
LibSetMathFcnRecArgExpr(FcnRec, idx, argStr) **9-51**

A model.rtw

model.rtw File Contents A-3
Using the model.rtw File **A-3**
General model.rtw Concepts **A-6**
Code Data Structure Concepts **A-7**
Inline Parameters On Code Structure **A-9**

model.rtw Changes Between
Real-Time Workshop 5.0 and 4.1 . . . . A-12

General Information and Solver Specification A-14

RTWGenSettings Record A-16

Data Logging Information (DataLoggingOpts) A-17

Data Structure Sizes A-19

Sample Time Information A-21

Data Type Information (DataTypes record) A-23

Target Properties A-25

Block Type Counts **A-26**

External Inputs and Outputs **A-27**

Block I/O Information **A-30**
Signal Connections **A-35**
Using BlockOutput[i].SigSrc **A-36**

Data Type Work (DWork) Information **A-42**

Continuous States **A-45**

Nonsampled Zero Crossings (NonsampledZCs) **A-46**

Zero Crossings Events (ZCEvents) **A-47**

Block Record Defaults **A-48**

Parameter Record Defaults **A-50**

Data and Control Port Defaults **A-51**

Model Parameters Record **A-54**

Reduced Blocks **A-58**

Custom Storage Class Record **A-59**

Model Hierarchy **A-60**

System Defaults
(canonical inputs, outputs, and parameters) **A-63**

System Record **A-65**

Stateflow Record **A-88**

Model Checksums **A-89**

Block Specific Records **A-90**
Subsystem Block Specific Records **A-112**
Linear Block Specific Records **A-115**

Object Records **A-118**

B 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-4**

TLC Error Messages **B-5**

TLC Function Library Error Messages **B-30**

C Using TLC with Emacs

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

Introducing the Target Language Compiler

The Target Language Compiler transforms a specially compiled Simulink block diagram into ANSI C files that can be compiled and linked manually or automatically by Real-Time Workshop. This initial discussion of Target Language Compiler technology includes the following topics.

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

Introduction

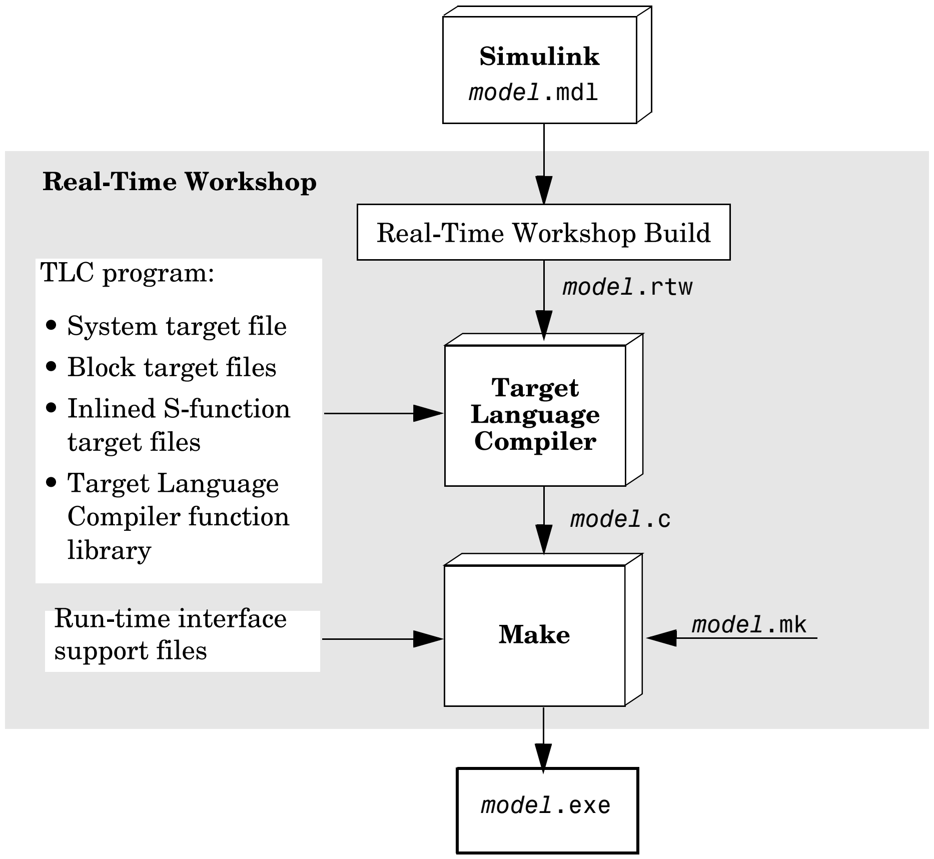
The Target Language Compiler™ tool is an integral part of the Real-Time Workshop®. It enables you to customize the C code generated from any Simulink® model and generate optimal, inlined code for your own Simulink blocks. Through customization, you can produce platform-specific code, or you can incorporate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

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

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.



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

As an integral component of Real-Time Workshop, the Target Language Compiler transforms an intermediate form of a Simulink block diagram, called `model.rtw`, into C code. The `model.rtw` file contains a "compiled" representation of the model describing the execution semantics of the block diagram in a very high level language. The format of this file is described in Chapter 5, "Contents of model.rtw."

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

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

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

Those familiar with HTML, Perl, and MATLAB® will find that the Target Language Compiler borrows ideas from each of them. It has mark-up syntax similar to HTML, along with the power and flexibility of Perl and other scripting languages, plus the data handling power of MATLAB (TLC can invoke MATLAB functions). The code generated by TLC is highly optimized and fully commented C code, and can be generated from any Simulink model, including linear, nonlinear, continuous, discrete, or hybrid. All Simulink blocks are automatically converted to code, with the exception of MATLAB function blocks and S-function blocks that invoke M-files. The Target Language Compiler uses *block target files* to transform each block in the `model.rtw` file and a *model-wide target file* for global customization of the code.

You can incorporate C MEX S-functions, along with the generated code, into the program executable. You can also write a target file for your C MEX S-function to *inline* the S-function (see "Inlining C MEX S-Functions" in Chapter 8), thus improving performance by eliminating function calls to the S-function itself and the memory overhead of the S-function's simStruct. Inlining an S-function incorporates the S-function block's code into the generated code for the model. When no TLC target file is present for the

S-function, its C code file is invoked via a function call. For more information on inlining S-functions, see Chapter 8, “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. For all other files created during code generation, including the `model.rtw` file, a build directory is used. This directory is created by Real-Time Workshop right in the current directory and is named `.model_target_rtw`, where `target` is the abbreviation for the target environment, e.g., `grt` is the abbreviation for the generic real-time target.

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

- The body for the generated C source code (`model.c`
- Header files (`model.h`
- Header file `model_private.h` defining parameters and data structures private to the generated code.
- A makefile, `model.mk`, for building the application.

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

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

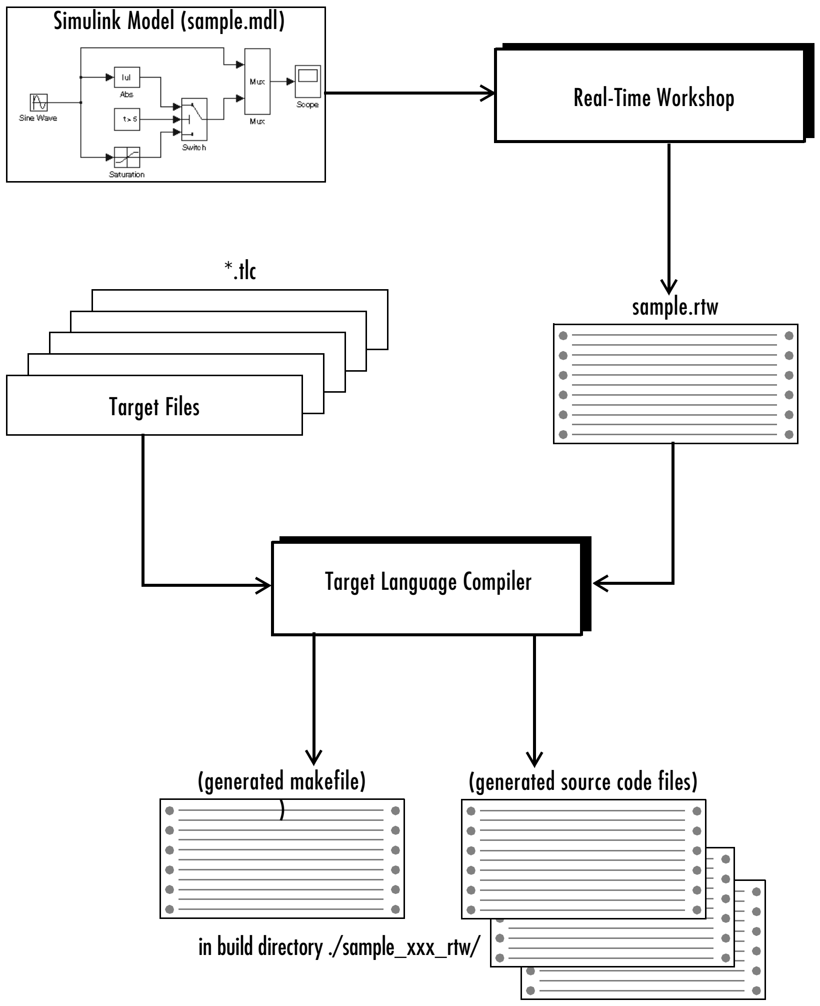


Figure 1-1: The Target Language Compiler Process

Capabilities

If you simply need to produce ANSI C code from Simulink models, you do not need to know how to prepare files for the Target Language Compiler. If you need to customize the output of Real-Time Workshop, you will need to instruct the Target Language Compiler. Use the 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 how blocks perform their functions, what datatypes are being manipulated, the structure of the `*model*.rtw` file, and how to modify target files to produce the desired output. Chapter 6, "Directives and Built-in Functions" describes the target language directives and their associated constructs. You will use the Target Language Compiler directives and constructs to modify existing target files or create new ones, depending on your needs. See "TLC Files" on page 4-19 for more information about target files.

Inlining S-Functions

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

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

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

Code Generation Process

Real-Time Workshop invokes the TLC after a Simulink model is compiled into an intermediate form (*model*.rtw) that is suitable for generating code. To generate code apropriately, the TLC uses its library of functions to transform two classes of target files:

- System target files
- Block target files

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

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

How TLC Determines S-Function Inlining Status

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

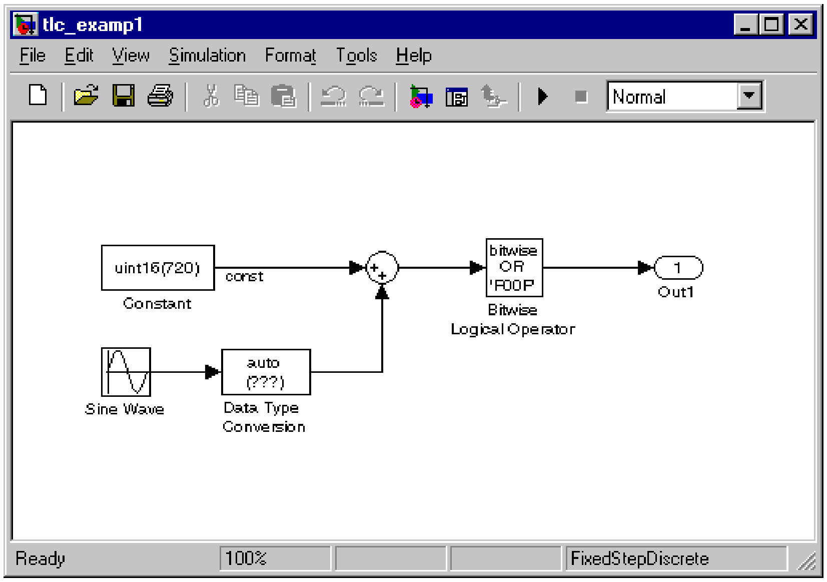
Because they cannot use simStructs, Fortran, and M-file S-functions must be inlined. This inlining can either be in the form of a full block target file or a “one-liner” block target file that references a “substitute” C MEX S-function source file.

A C MEX S-function will be selected for inlining by the Target Language Compiler if there is an explicit `mdlRTW()` function in the S-function code or if there is a target file for the current target language for the current block in the TLC file search path. If a C MEX S-function has an explicit `mdlRTW()` function, there must be a corresponding target file or an error condition will result.

The target file for an S-function must have the same root name as the S-function and must have the extension `.tlc`. For example, the example C MEX S-function source file `sfun_bitop.c` has its compiled form in `toolbox/simulink/blocks/sfun_bitop.dll` (`.mex*` for UNIX) and its C target file is located in `toolbox/simulink/blocks/tlc_c/sfun_bitop.tlc`.

A Look at Inlined and Noninlined S-Function Code

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



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

```
/* Bitwise Logic Block: <Root>/Bitwise Logical Operator */
/* [input] OR 'F00F' */
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 `mdlOutputs` function of the S-function, code would be generated like this:

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

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

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

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

      {
        int_T i;

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

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

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

        /* port 0 */
        {

          static real_T const *sfcnUPtrs[1];
          sfcnUPtrs[0] = &rtU.In1;
          ssSetInputPortSignalPtrs(rts, 0, (InputPtrsType)&sfcnUPtrs[0]);
          _ssSetInputPortNumDimensions(rts, 0, 1);
          ssSetInputPortWidth(rts, 0, 1);
        }
```

```
          }
    .
    .
    .
```

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

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

Advantages of Inlining S-Functions

Motivations

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

While switching algorithms is a desirable feature in the design phase of a system, when the time comes to generate code, this type of flexibility is often dropped in favor of optimal calculation speed and code size. 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 data type
- A significant or large code size in the `mdlOutputs()` function
- Multiple instances of this block in your models

Whenever you encounter this situation, the effort of inlining the block may not improve execution speed and could actually increase the size of the generated code. The trade-off is in the size of the block's body code generated for each instance vs. the size of the child SimStruct created for each instance of a noninlined S-function in the generated code.

Alternatively, you can use a hybrid inlining method known as a C MEX wrappered 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 8, “Inlining S-Functions” for the procedure and an example of a wrappered 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 `./tlc_c` (for C language targets)

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

Availability for Inlining and Noninlining

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

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

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

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

New Features Added in Version 5.0

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

- A C-like `SPRINTF` built-in formatting function has been added which returns a TLC string encoded with data from a variable number of arguments.
- `BlockInstanceData` function has been depreciated

  S-functions should no longer call the `BlockInstanceData` function. All data used by a block should be declared using data type work vectors (DWORK).
- Unified Code Generation for Real-Time Workshop and Stateflow

  In earlier releases, code generated from Stateflow charts in a model was written to source code files distinct from the source code files (such as model.c, model.h, etc.) generated from the rest of the model. Now, by default, Stateflow no longer generates any separate files from the Real-Time Workshop. For example, all Stateflow initialization code is now inlined.
- A new directive, `%filescope`, can be used to limit the scopes of variables to the files they are defined in. All variables defined after the appearance of `%filescope` in a file will have this property, otherwise they will default to global variables.
- Use of the `::` operator to access global variables is now allowed in TLC files. Variables defined on the command line and records read from *`model`*.rtw files will remain global variables. Nested include files cannot access variables local to the file which included them.
- The `%assert` directive (which tests the value of a Boolean expression and issues an error message, prints a stack trace, and then exits if the result is `FALSE`) is now easier to control.

  You may enable/disable such `%assert` tests in several ways: via the `-da` command line switch, by the `%setcommandswitch` directive within files, using the `set_param(model, 'TLCAssertion', 'on|off')` command, or with a checkbox control on the Real-Time Workshop GUI. By default, the checkbox is empty (`%assert` directives are ignored).

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

New Features Added in Version 4.1

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

- The TLC Debugger is now supported. See Chapter 7, "Debugging TLC Files."
- `ISINF`, `ISNAN` and `ISFINITE` now work for complex values.
- Added support for literal strings.

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

  The following examples are equivalent.

  - `L"d:\this\is\a\path"`
  - `"d:\\this\\is\\a\\path"`
- Zero indexing for complex values is now supported, as in:

  ```
  %assign a = 1.0 + 3.0i
  %assign b = a[0] %% this didn't work before
  ```
- The following new functions have been added to the TLC Function Library:
  - `LibBlockInputSignalConnected`
  - `LibBlockInputSignalLocalSampleTimeIndex`
  - `LibBlockInputSignalOffsetTime`
  - `LibBlockInputSignalSampleTime`
  - `LibBlockInputSignalSampleTimeIndex`
  - `LibBlockOutputSignalOffsetTime`
  - `LibBlockOutputSignalSampleTime`
  - `LibBlockOutputSignalSampleTimeIndex`
  - `LibBlockMatrixParameterBaseAddr`

- `LibBlockParameterBaseAddr`
- `LibBlockNonSampledZC`

See Chapter 8, “Inlining S-Functions” for information on these functions.

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

New Features Added in Version 4.0

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

- Complete parsing of the TLC file just before execution. This aids development because syntax errors are caught the first time the TLC file is run instead of the first time the offending line is reached.
- TLC speed improvements across the board, particularly in block parameter generation.
- Creation and use of a build directory in the current directory to prevent generated code from clashing with other files generated for other targets, and for keeping your model directories maintenance to a minimum.
- Entirely new TLC Profiler for finding performance problems in your TLC code.
- New format and changes to the *`model`*`.rtw` file. 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 *`ParamName`*`0` in Appendix A, “model.rtw,” for details.
- New flexible methods for text expansion from within strings.
- Column-major ordering of 2-dimensional signal and parameter data.
- `FIELDNAMES`, `GENERATE_FORMATTED_VALUE`, `GETFIELD`, `ISALIAS`, `ISEMPTY`, `ISEQUAL`, `ISFIELD`, `REMOVEFIELD`, `SETFIELD`Support for 2-dimensional signals in inlined code.
- `INTMAX, INTMIN, TLC_TRUE, TLC_FALSE, UINTMAX`
- Functions can return records.
- Formalization of records and record aliases.
- Loop control variables are local to loop bodies.

- Improved `EXISTS` semantics (See “Built-In Functions and Values” on page 6-39).
- Can expand records with `%<>`
- Short circuiting of conditionals (`||, &&, ?:, %if-%elseif-%else-%endif`)
- 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

Compatibility Issues in Version 5.0

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

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

  Designed to reduce code size and improve readability, the rtModel is a lightweight structure that is dynamically created when compiling a model, containing only those fields required to execute that model. GRT now utilizes the SimStruct only for noninlined child S-functions.
- Changes to the format of the *`model`*`.rtw` file may require you to update TLC files that access *`model`*`.rtw` records, especially if they do so directly rather than by calls to the TLC function library.

Compatibility Issues in Version 4.1

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

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

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

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

Compatibility Issues in Version 4.0

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

- Nested evaluations are no longer supported. Expressions such as

  ```
  %<LibBlockParameterValue(%<myVariable>,"", "", "")>
  ```

  are no longer supported. You will have to convert these expressions into equivalent nonnested expressions.

- Aliases are no longer automatically created for Parameter blocks while reading in Real-Time Workshop files.
- You cannot change the contents of a “Default” record after it has been created. In the previous TLC, you could change a “Default” record and see the change in all the records that inherited from that default record.
- `%codeblock` and `%endcodeblock` constructs are no longer supported.
- `%defines` & macro constructs are no longer supported.
- Use of line continuation characters (`...` and `\`) are not allowed inside of strings. Also, to place a double quote (`"`) character inside a string, you must use `\"`. Previously, TLC allowed you to do `"""` to get a double quote in a string.
- Semantics have been formalized to `%include` files in different contexts (e.g., from generated files inside of `%with` blocks, etc.) `%include` statements are now treated as if 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 `%function` in one file and `%include` a file that had the `%endfunction`). This no longer works.

- Nested functions are no longer allowed. For example,

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

- Recursive records are no longer allowed. For example,

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

- Record declaration syntax has changed. The following code 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 4-14 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, “A TLC Tutorial,” contains a selection of examples with step-by-step instructions on how to use the TLC.
- Chapter 4, “Code Generation Architecture,” describes the TLC files and the build process. It also provides a tutorial on how to write target language files.
- Chapter 5, “Contents of model.rtw,” describes the `model.rtw` file.
- Chapter 6, “Directives and Built-in Functions,” contains the language syntax for the Target Language Compiler.
- Chapter 7, “Debugging TLC Files,”explains how to use the TLC debugger.
- Chapter 8, “Inlining S-Functions,” describes how to use the Target Language Compiler and how to inline S-functions.
- 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 documentation 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 “Targeting Real-Time Systems” chapter offers further useful examples and customization guidelines.

- The Real-Time Workshop Embedded Coder documentation presents details on generating code for embedded targets. Among other topics, it covers data structures and program execution, code generation, custom storage classes, module packaging, and specifies system requirements and restrictions on target files.
- The Simulink 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 wrappered and fully inlined S-functions with a special emphasis on the `mdlRTW()` function.

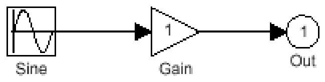
Getting Started

The following sections outline the steps involved in converting *`model`*`.rtw` files to working source code, and focus on the most common application of customization using TLC, inlining S-functions.

Code Architecture (p. 2-2)	What information code for a block captures
model.rtw and Target Language Compiler Overview (p. 2-4)	How the Target Language Compiler interprets *`model`*`.rtw` files
Inlining S-Function Concepts (p. 2-6)	Techniques used for inlining, with examples

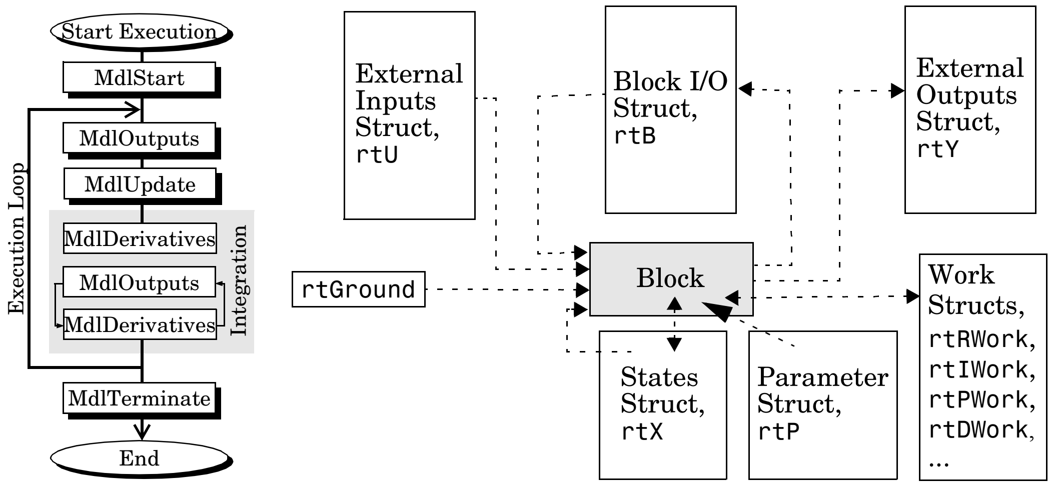
Code Architecture

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



```
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;

  /* Outport 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 Target Language Compiler-specific pieces of the code generation process.

model.rtw and Target Language Compiler Overview

The Target Language Compiler Process

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

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

model.rtw Structure

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

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

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

```
CompiledModel {
  System {
    Block {
      DataInputPort {
        ...
      }
      DataOutputPort{
        ...
      }
      ParamSettings {
        ...
      }
      Parameter {
        ...
      }
    }
  }
}
```

Operating Sequence

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

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

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

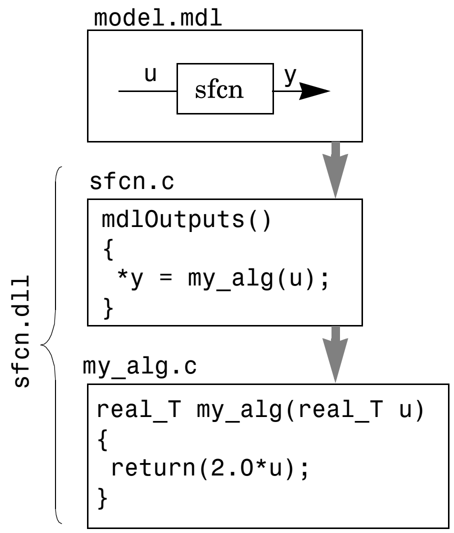
Inlining S-Function Concepts

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

Noninlined S-Function

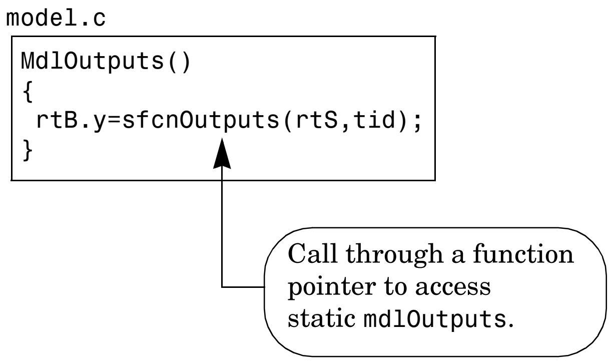
If an inlining TLC file is not provided, most Real-Time Workshop targets will still support the block by recompiling the C-MEX S-function for the block. As discussed earlier, there is overhead in memory usage and speed when using the C coded S-function and only a limited subset of `mx*` API calls are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

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



sfcn.c
```
mdlOutputs()
{
 *y = my_alg(u);
}
```

my_alg.c
```
real_T my_alg(real_T u)
{
 return(2.0*u);
}
```


model.c
MdlOutputs()
{
 rtB.y=sfcnOutputs(rtS,tid);
}
Call through a function pointer to access static mdlOutputs.

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

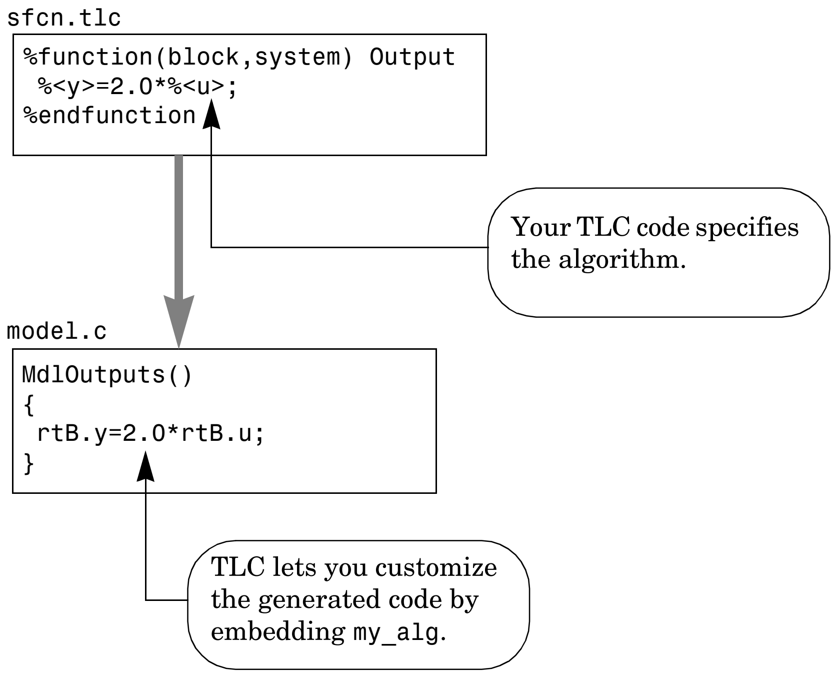
While both effectively inline the S-function and remove the overhead of a noninlined S-function, the two approaches are different. The first example below using `timestwo.tlc` is considered a fully inlined TLC file, where the full implementation of the block is contained in the TLC file for the block.

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 opposed to each creating identical algorithmic code.
- It provides a way to incorporate legacy C code seamlessly into the Real-Time Workshop generated code.

Fully Inlined S-Function Example

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


sfcn.tlc
%function(block,system) Output
%<y>=2.0*%<u>;
%endfunction
Your TLC code specifies the algorithm.
model.c
MdlOutputs()
{
rtB.y=2.0*rtB.u;
}
TLC lets you customize the generated code by embedding my_alg.

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

timestwo.tlc

```
%implements "timestwo" "C"

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

TLC Block Analysis

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

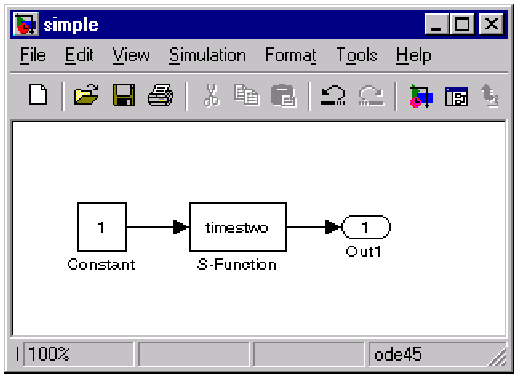
The last piece to the prototype is `Output`. This means that any line that is not a TLC directive is output by the function to the current file that is selected in TLC. So, any nondirective lines in the `Outputs` function become generated code for the block.

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

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

timestwo Model

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

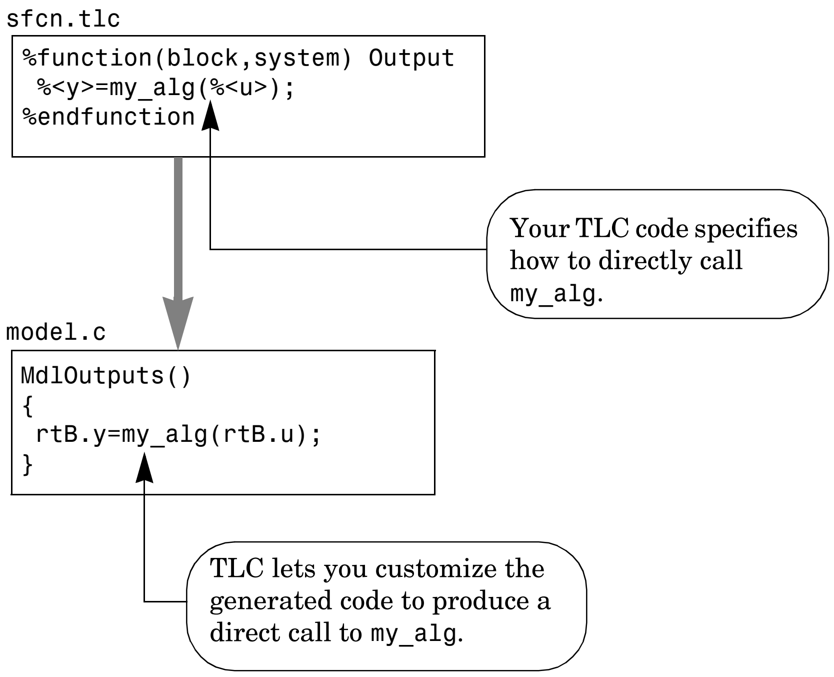


MdlOutputs Code

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

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

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.


sfcn.tlc
%function(block,system) Output
%<y>=my_alg(%<u>);
%endfunction
Your TLC code specifies how to directly call my_alg.
model.c
MdlOutputs()
{
rtB.y=my_alg(rtB.u);
}
TLC lets you customize the generated code to produce a direct call to my_alg.

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

```
%implements "timestwo" "C"

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

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

%endfunction
```

Analysis

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

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

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

  /* Outport 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 5, “Contents of model.rtw,” and Appendix A, “model.rtw,” contain more details of the *`model`*`.rtw` file and its contents. Chapter 8, “Inlining S-Functions,” and Chapter 3, “A TLC Tutorial,” contain details on writing TLC files, including a comprehensive tutorial.

A TLC Tutorial

The fastest and easiest way to understand the Target Language Compiler is to run it, paying attention to how TLC scripts transform compiled Simulink models (`model.rtw` files) into source code. The exercises in the following tutorial are designed to highlight the principal reasons for and techniques of using TLC:

Introduction (p. 3-2)	About the TLC Tutorials
Reading Record Files with TLC (p. 3-3)	How record files are structured, and how TLC accesses them and interprets their contents
Processing model.rtw Files with TLC Scripts (p. 3-18)	Exploring how TLC library functions extract data from `model.rtw` files
Inlining S-Functions with TLC (p. 3-25)	The basics of inlining S-functions, the most common application of TLC customization
Exploring Variable Names and Loop Rolling (p. 3-32)	Understanding how and when TLC outputs loops, and how variable names are affected
Debugging Your TLC Code (p. 3-40)	Essential TLC debugger commands, setting breakpoints, and tracing execution
Using TLC Code Coverage to Aid Debugging (p. 3-49)	A simple approach to getting information about which TLC statements execute
Wrapping User Code with TLC (p. 3-53)	Using TLC to integrate legacy and proprietary code into compiled models

Introduction

This set of tutorials provides a number of Target Language Compiler (TLC) exercises, each one organized as a major section.

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

Note Before you begin the tutorial, copy the entire `tlctutorial` directory to a local working directory. All the files you'll need will be together, and if you make mistakes or want fresh examples to try again, you can always recopy files from *`matlabroot`*`/`.

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

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

- Working in the MATLAB environment
- Building models in Simulink
- Using Real-Time Workshop to generate code for target systems
- High-level language concepts (e.g., C or Fortran programming)

If you encounter terms in the tutorials that you don't understand, it may be helpful to review "Code Generation Concepts" on page 4-13 to acquaint yourself with the basic goals and methods of TLC programming. Similarly, if you see TLC keywords, built-in functions, or directives that you would like to know more about, please see Chapter 6, "Directives and Built-in Functions." Descriptions of TLC library functions are provided in Chapter 9, "TLC Function Library Reference."

Reading Record Files with TLC

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

Example directory: `tlctutorial/guide`

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

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

- "Structure of Record Files" — Some background and a simple example
- "Interpreting Records" — Presenting contents of the record file
- "Anatomy of a TLC Script" — Deconstructing the presentation
- "Modify read-guide.tlc" — Have some fun with TLC

Structure of Record Files

Real-Time Workshop compiles models into a structured form called a *record file*, referred to as *`model`*`.rtw`. Such compiled model files are similar in syntax and organization to source model (`model.mdl`) files, in that they contain a series of (usually hierarchically nested) records of the form

```
recordName {itemName itemValue}
```

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

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

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

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

As long as the programmer knows the names of records and fields, and their expected contents, he or she can compose TLC statements to read, parse, and manipulate record file data.

Interpreting Records

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

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

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

* Concatenate string values, e.g.
  "%assign worker = worker + " " + Top.Employee.LastName"
  worker <- worker + " " + Top.Employee.LastName = "Arthur Dent"
```

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

* Put variables into a field, e.g.
  "%assign Top.Employee.GrossRate = wageCost"
  Top.Employee.GrossRate <- wageCost = 20.47

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

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

This "presentation" of `guide.rtw` was produced by invoking the Target Language Compiler from the MATLAB command window, executing a script called `read-guide.tlc`. Do this yourself now, by following these steps:

1 If you haven't already done so, copy the directory `tlctutorial/guide` to your working directory.

2 In MATLAB, `cd` to `/guide` within your working directory

3 To produce the output just listed, process `guide.rtw` with the TLC script `read-guide.tlc` by typing the following command:

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

Note command usage:

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

Anatomy of a TLC Script

Let's dissect the script we just ran. Each "paragraph" of output from `guide.tlc` is discussed in sequence in the following brief sections:

- "Coding Conventions" — Before you begin
- "File Header" — Header info and a formatting directive
- "Token Expansion" — Evaluating field and variable identifiers
- "General Assignment" — Using the `%assign` directive
- "String Processing Plus" — Methods of assembling strings
- "Arithmetic Operations" — Computations on fields and variables
- "Modifying Records" — Changing, copying, appending to records
- "Traversing Lists" — Referencing list elements with subscripts
- "Looping over Lists" — Details on loop construction and behavior

Coding Conventions

These are some basic TLC syntax and coding conventions.

`%% Comment`	TLC comment, which will not be output
`/* comment */`	Comment, to be output
`%keyword`	TLC directives start with "%"
. (period)	Scoping operator, e.g., `Top.Lev2.Lev3`
`...` (at end-of-line)	Statement continuation (no line break output)
`\` (at end-of-line)	Statement continuation (line break is output)
`localvarIdentifier`	Local variables start in lowercase
`GlobalvarIdentifier`	Global variables start in uppercase
`RecordIdentifier`	Record identifiers start in uppercase
`EXISTS()`	TLC built-in functions are named in uppercase *Note: All TLC identifiers are case-sensitive.*

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

File Header

`read-guide.tlc` begins with

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

The first three lines are comments. All text on a line following the characters "`%%`" is treated as a comment (ignored, not interpreted or output).

The fourth line, as explained in the third line, is the TLC directive (keyword) `%realformat`, which controls how subsequent floating-point numbers will be formatted when output. Here we want to minimize the digits displayed.

Token Expansion

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

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

The first two lines (and any line that contains no TLC directives or tokens) are simply echoed to the output stream, including leading and trailing spaces.

The third and fourth lines will be explained momentarily.

The last line evaluates (expands) the record `Top.Date`. More precisely, it evaluates the field called `Date` existing in the scope called `Top`. The syntax `%<expr>` causes expression `expr` (which can be a record, a variable, or a function) to be evaluated. This operation is sometimes referred to as an *eval*.

Note The `%<>` operator cannot be nested (i.e., `%<foo%<bar>>` is not allowed).

The third line creates a variable named `td` and assigns a string value to it. The `%assign` directive creates new and modifies existing variables. Its general syntax is:

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

The optional double colon prefix specifies that the variable being assigned to is a global variable. In its absence, a local variable in the current scope is created or modified.

The fourth line prints out as

```
%<Top.Date> -- evaluates to:
```

In order to enable the TLC to print “`%<Top.Date>`” without expanding it beforehand, the preceding line was needed to construct that string by pasting together two literals:

```
%assign td = "%" + "<Top.Date>"
```

As discussed under “String Processing Plus” below, the plus operator concatenates strings as well as adds numbers.

General Assignment

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

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

The first line is simply echoed to output. The second line is an assignment of a field called `FirstName` in the `Top.Employee` record scope to a new local variable called `worker`.

As the second line does not produce output, it was necessary to add a line to show what it is. The third line, therefore, repeats the previous statement, making it visible by enclosing it in quotation marks.

The fourth line then indicates what the assignment caused to happen, and illustrates the token expansion that took place.

Note When the `%<>` operator is used within quotation marks, e.g. “`%<Worker>`”, the TLC will expand the expression, and then enclose the result in quotation marks. However, placing `%assign` within quotation marks, e.g. “`%assign foo = 3`”, simply echoes the statement, enclosed in quotation marks, to the output stream. No assignment will result (the value of `foo` remains unchanged or undefined)

String Processing Plus

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

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

The second line performs the concatenation, the third line echoes it, and the fourth line describes the operation, in which a variable is concatenated to a field separated by a space character. An alternative way to do this, without using the + operator, is

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

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

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

- Numeric Types — Add the two expressions together; both operands must be numeric.
- Vectors — If the first argument is a vector and the second is a scalar, the scalar is appended to the vector.
- Matrices — If the first argument is a matrix and the second is a vector of the same column-width as the matrix, the vector is appended as another row to the matrix.
- Records — If the first argument is a record, the second argument is added as a parameter identifier (with its current value).

Arithmetic Operations

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

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

Again, the second line is the `%assign` statement that computes the value, which is stored in local variable `wageCost`, and the third line echoes the operation. Note that the fourth and fifth lines compose a single statement. The ellipsis

("`...`", typed as three consecutive periods) signals that a statement is continued on the following line, but if the statement has output no line break will be inserted. To continue a statement *and* insert a line break, use a backslash ("`\`") in place of "`...`".

Modifying Records

Once read into memory, records in record files can be modified and manipulated just like variables created by assignment. The next segment of `read-guide.tlc` replaces the value of record field `Top.Employee.GrossRate`:

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

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

Several TLC directives beside `%assign` may be used to modify records:

• `%createrecord`	Creates new top-level records, and may also specify subrecords within them, including name/value pairs.
• `%addtorecord`	Adds fields to an existing record. The new fields may be name/value pairs or aliases to existing records.
• `%mergerecord`	Combines one or more records. The first record will contain itself plus copies of the contents of all the other records specified by the command, in sequence.
• `%copyrecord`	Creates a new record like `%createrecord` does, except the components of the record come from the existing record specified.
• `%undef` *`var`*	Removes (deletes) *`var`* (a variable or a record) from scope. If `var` is a field in a record, field is removed from the record. If `var` is a record array (list), removes the first element of the array; remaining elements remain accessible. Only records created via `%createrecord` or `%copyrecord` can be removed.

See “Compiler Directives” on page 6-2 for further details on these related directives.

Traversing Lists

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

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

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

While explicit indexing such as the above is perfectly acceptable, it is often preferable to use a loop construct when traversing entire lists, as shown next.

Looping over Lists

By convention, the section of a record file that a list occupies is preceded by a record that indicates how many list elements are present. In *`model`*`.rtw` files such parameters are declared as `NumIdent`, where `Ident` is the identifier used for records in the list that follows. In `guide.rtw`, the `Project` list looks like this:

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

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

Note *model*.rtw files contain records which start with "Num" but which are *not* list size parameters. TLC itself does not require that list size parameters start with Num. Programmers therefore need to be cautious when interpreting Num– record identifiers. The built-in TLC function SIZE() can determine the number of records in a specified scope, hence the length of a list.

The last segment of read-guide.tlc uses a %foreach loop, controlled by the NumProject parameter, to iterate the Project list and manipulate its values. As you recall, the TLC output looks like this:

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

The TLC statements which generated this output are

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

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

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

Modify read-guide.tlc

Now that you have studied `read-guide.tlc`, it's time to get under the hood and tinker with it. In this hands-on exercise, we introduce two important TLC facilities, *file control* and *scoping control*. Your challenge is to implement both within the `read-guide.tlc` script.

File Control Basics

TLC scripts almost invariably produce output in the form of streams of characters. Outputs are normally directed to one or more buffers and files (a buffer is a volatile file stored in RAM), collectively called *streams*. So far, output from `read-guide.tlc` has been directed to the MATLAB command window, because you told it to do so by including the `-v` switch on the command line. Prove this by omitting `-v` when you run `read-guide.tlc`. Type

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

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

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

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

If no `filename` is given, subsequent output flows to the memory buffer named by `streamid`. If no `mode` is specified, the file is opened for writing, and any existing content in it will be deleted (subject to system-level file protection mechanisms). Valid `mode` identifiers are "`a`" (append), "`r`" (readonly, seldom used) and "`w`" (write, the default). Enclose these characters in quotes.

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

The default output stream (which you can respecify with streamid `NULL_FILE`) is null. Another built-in stream is `STDOUT`. When activated using `%selectfile`, `STDOUT` directs output to the MATLAB command window.

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

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

File When Ready

In your working `/guide` directory is a file called `read-guide-file-src.tlc`, a version of `read-guide.tlc` with comments and three lines of text added. Edit this file to implement output file control, as follows:

1. Open `read-guide-file-src.tlc` in your text editor.
2. Note five comment lines that commence with "`%% ->`".

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

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

   If you succeeded, a file called `guidetext.txt` will be created containing the expected output, and the command window will display:

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

If you did not see these messages, or if no text file was produced, review the material and try again. If problems persist, inspect `read-guide-file.tlc` in the `/solutions` subdirectory to see how you should have specified file control.

The Scoop on Scopes

The “Structure of Record Files” on page 3-3 explained the hierarchical organization of records. Each record exists within a scope defined by the record(s) in which it is nested. Our example file, `guide.rtw`, contains the following scopes:

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

To refer to a field or a record, it is normally necessary to specify its scoping, even if there is no other context that contains the identifier. For example, in `guide.rtw` the field `FirstName` exists only in the scope `Top.Employee`, yet it must be referred to as `Top.Employee.FirstName` whenever it is accessed.

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

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

is a scope in the `f14a.rtw` file used in a later exercise.

The `%with`/`%endwith` directive eases the burden of correctly coding TLC scripts and to clarify their flow of control. Syntax is

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

Every `%with` is eventually followed by an `%endwith`, and these pairs may be nested (but not overlapping). If `RecordName` is below the top level, the top-level scope need not be included in its description. For example, to make the current scope of a *`model`*`.rtw` file `CompiledModel.DataTypes`, you can specify

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

Naturally, “`%with CompiledModel.DataTypes` is also valid syntax. Once bracketed by `%with`/`%endwith`, record identifiers in TLC statements no longer require their outer scope to be specified. However, two conditions should be noted:

- Records outside of the current `%with` scope can be accessed, but must be fully qualified, as usual.
- Whenever assignments to records are made inside a `%with` directive (i.e., they are produced as lvalues), their names must be fully qualified.

Get %with It

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

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

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

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

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

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

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

The next exercise, "Processing model.rtw Files with TLC Scripts,"also uses `%with ... %endwith` scoping, in the context of processing an actual *`model`*`.rtw` file.

For additional information, see “Using Scopes in the model.rtw File” on page 5-3 and “Variable Scoping” on page 6-55 in this document.

Processing model.rtw Files with TLC Scripts

Objective: Learn to create and modify the behavior of Target Language Compiler files that use built-in and library functions and to run them from command lines.

Example directory: `tlctutorial/listrtw`

In this exercise you modify a TLC script to summarize the contents of an RTW model, a version of the f14 aircraft model that is supplied as a Simulink demo. Remember to work with a copy of this directory; never modify the original files.

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

- "Getting Started" — Setting up the example files
- "Counting Blocks and Subsystems"— Summarizing a *`model`*`.rtw` file
- "Listing Block Names" — Expanding the summary
- "Passing and Using a Parameter"— Controlling summarization

Note From time to time, The MathWorks may change the format of the *`model`*`.rtw` file in minor ways or restructure TLC libraries. As a result, user-written programs such as the one presented in this exercise may require small modifications to continue to work correctly.

Getting Started

Use the file called `f14a.rtw` in the `listrtw` subdirectory. This differs from the Simulink f14 demo in that its four subsystems have been designated as atomic. The original demo `f14.mdl` can be run by typing `f14` at the MATLAB prompt. It is stored in *`matlabroot`*`/toolbox/simulink/simdemos/aerospace` directory.

Note You may designate subsystems to be atomic units via the **Subsystem parameters...** dialog box summoned by right-clicking on a subsystem mask. All blocks within atomic subsystems are executed as a unit. Conditionally executed subsystems are atomic, while virtual subsystems are not.

You will also find a file called `listrtw1.tlc`. This is the file you will modify for this exercise. After you complete an exercise, you can consult files in the `tlctutorial/listrtw/solutions` subdirectory to check your results.

Counting Blocks and Subsystems

The Target Language Compiler is a program that transforms *`model`*`.rtw` record files and S-functions into C source code. In this exercise you will modify a program that reports the blocks contained in a *`model`*`.rtw` file. You can use the TLC utility scripts from this exercise to list the contents of any such `.rtw` file, not just the one used in the example.

1 Open `listrtw1.tlc` in your text editor. As provided, it contains

```
%% File: listrtw1.tlc
%% Count blocks and subsystems of a model.rtw file
%%
%%
%% NOTE: Change "<matlabroot>/" below to the
%%       MATLAB main directory on your system
%addincludepath "<matlabroot>//rtw//c//tlc//lib"
%assign Accelerator = 0     %%Needed to avoid error in utillib
%include "utillib.tlc"
%selectfile STDOUT

*** SYSTEMS AND BLOCKS IN RECORDFILE
%assign nbls = 0
%with CompiledModel
    %foreach sysIdx = NumSystems
        %with System[sysIdx]
            %assign nbls = nbls + NumBlocks
*** %<NumBlocks> blocks in system %<sysIdx + 1>
        %endwith
    %endforeach
*** recordfile contains %<nbls> blocks in %<NumSystems> systems
%endwith
*** END LISTING

%% end listrtw1.tlc
```

Note that the following TLC directives are used (refer to Chapter 6, “Directives and Built-in Functions,” for detailed descriptions):

`%addincludepath`	Enables TLC to find included files. You must edit this string to describe the MATLAB root directory. Notice that the `‘/’` character has to be escaped (doubled).
`%assign`	Creates or modifies a variable
`%include`	Inserts one file in another, as in C
`%selectfile`	Directs output to a stream or file
`%with...%endwith`	Adds a scope to simplify referencing blocks
`%foreach...%endforeach`	Iterates loop variable from 0 to upper limit

2. Find the line `%addincludepath "<matlabroot>//rtw//c//tlc"` and change `<matlabroot>` to specify the MATLAB root directory on your system. The double slashes are required to parse the string properly.

3. Run `listrtw1.tlc` on the `f14a.rtw` file. Do this by typing the `tlc` command in MATLAB (type `help tlc` for command syntax information). In the MATLAB command window (having made sure you are currently in your own `/listrtw` directory) type

   ```
   tlc -r f14a.rtw listrtw1.tlc
   ```

The model file is read and analyzed by `listrtw1.tlc`, which prints the following summary of it in the MATLAB command window:

```
*** SYSTEMS AND BLOCKS IN RECORDFILE
*** 8 blocks in system 1
*** 10 blocks in system 2
*** 4 blocks in system 3
*** 7 blocks in system 4
*** 17 blocks in system 5
*** recordfile contains 46 blocks in 5 systems
*** END LISTING
```

Listing Block Names

Because you might want to itemize a model as well as to count how many blocks it contains, let's add TLC code that causes the name of the input file and the names of its blocks to be printed in the listing. To obtain the input filename, we use a built-in TLC function. For referencing block names, we invoke a TLC library function:

1. Open `listrtw1.tlc` in your editor and save it as `listrtw2.tlc`. Be sure to change the argument to `addincludepath` to point to *`matlabroot`*`/` on your system.

2. To enable your program to access the input filename from the command line, do the following three things:

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

      The `%assign` directive declares and sets variables. In this instance, it holds a string filename identifier. `GET_COMMAND_SWITCH()` returns whatever string argument follows a specified TLC command switch. Note that uppercase is always used for built-in function names.

   b. Change the line "`*** SYSTEMS AND BLOCKS IN RECORDFILE`" to read as follows:
   ```
   *** SYSTEMS AND BLOCKS IN RECORDFILE %<inputfile>
   ```

   c. Change the line "`*** recordfile contains %<nbls> blocks in %<NumSystems> systems`" to:
   ```
   *** recordfile %<inputfile> contains %<nbls> blocks in ...
   %<NumSystems> systems
   ```
      The "`%< >`" operator causes variables to be evaluated, substituting their contents for the tagged expression. The ellipsis ("`...`") at the end of the line indicates that the line is continued below, but prints without a line break.

3. To report the name of each block as it is iterated:

   Below the line "`%assign nbl = nbl + 1`" add a new statement:

```
%<LibGetFormattedBlockPath(Block[blkIdx])>
```

The TLC library function `LibGetFormattedBlockPath` returns the full pathname string of a block without any special characters suitable for displaying the block name on a single line. TLC substitutes the returned value for the bracketed expression. Source code for the function is found in `utillib.tlc`, which is that library had to be included in the TLC script.

4 Save `listrtw2.tlc` and execute it with

```
tlc -r f14a.rtw listrtw2.tlc
```

The output is a report of block names grouped by subsystem (see listing in the next section, "Passing and Using a Parameter").

Passing and Using a Parameter

As we described above, and as you demonstrated by using the built-in function `GET_COMMAND_SWITCH()`, you can use the MATLAB `tlc` command to pass parameters from the command line to the TLC file being executed. The most general command switch is `-a`, which assigns arbitrary variables. For example:

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

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

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

Note that such variables need not be declared in the TLC file, and are available for use when set with `-a`. However, errors result if the code assigns undeclared variables that are not specified by an `-a` switch when invoking the file. Also note that (in contrast to the `-r` switch) no space should separate `-a` from the parameter being declared.

In the final section of this exercise, we provide a parameter to control whether or not the names of blocks are printed out. By default block names are listed, but are suppressed if the command line contains `-alist=0`.

1 Open `listrtw2.tlc` in your editor and save it as `listrtw3.tlc`

Below the line `%include "utillib.tlc"` you need to check whether a `list` parameter has been declared, via the intrinsic (built-in) function `EXISTS()`. Should no `list` variable exist, your program assigns one:

```
%include "utillib.tlc"
%if (!EXISTS(list))
   %assign list = 1
%endif
```

This code ensures that `list` is defined and by default its value is `TRUE`.

2 Next, enclose the line that prints block names within an `%if` block:

```
%if (list)
    %<LibGetFormattedBlockPath(Block[blkIdx])>
%endif
```

Now `LibGetFormattedBlockPath` is called and expanded into text only when `list` is `TRUE`.

3 Save `listrtw3.tlc` and cverify that your changes work as expected, using the command

```
tlc -r f14a.rtw -alist=0 listrtw3.tlc
```

With the `-alist=0` switch, the output resembles that from `listrtw1.tlc`, except that the input filename is reported as well as the block and subsystem counts.

4 Finally, execute `listrtw3.tlc` with the command:

```
tlc -r f14a.rtw listrtw3.tlc
```

This yields the same results as you obtained with `listrtw2.tlc`. Here is the output printed to the MATLAB command window:

```
*** SYSTEMS AND BLOCKS IN RECORDFILE [f14a.rtw]
    f14a/Aircraft Dynamics Model/Transfer Fcn.1
    f14a/Aircraft Dynamics Model/Gain3
    f14a/Aircraft Dynamics Model/Transfer Fcn.2
    f14a/Aircraft Dynamics Model/Gain4
    f14a/Aircraft Dynamics Model/Gain5
    f14a/Aircraft Dynamics Model/Gain6
    f14a/Aircraft Dynamics Model/Sum1
    f14a/Aircraft Dynamics Model/Sum2
*** 8 blocks in system 1
    f14a/Controller/Alpha-sensor Low-pass Filter
    f14a/Controller/Stick Prefilter
    f14a/Controller/Pitch Rate Lead Filter
    f14a/Controller/Gain2
    f14a/Controller/Gain3
    f14a/Controller/Sum1
    f14a/Controller/Sum2
    f14a/Controller/Gain
    f14a/Controller/Proportional plus integral compensator
    f14a/Controller/Sum
*** 10 blocks in system 2
    f14a/Dryden Wind Gust Models/Band-Limited White Noise/White Noise
    f14a/Dryden Wind Gust Models/Band-Limited White Noise/Gain
    f14a/Dryden Wind Gust Models/W-gust model
    f14a/Dryden Wind Gust Models/Q-gust model
*** 4 blocks in system 3
    f14a/Nz pilot calculation/Constant
    f14a/Nz pilot calculation/Derivative
    f14a/Nz pilot calculation/Derivative1
    f14a/Nz pilot calculation/Gain1
    f14a/Nz pilot calculation/Product
    f14a/Nz pilot calculation/Sum1
    f14a/Nz pilot calculation/Gain2
*** 7 blocks in system 4
    f14a/Actuator Model
    f14a/Dryden Wind Gust Models
    f14a/Gain
    f14a/Gain2
    f14a/Gain1
    f14a/Sum
    f14a/Aircraft Dynamics Model
    f14a/Gain5
    f14a/alpha (rad)
    f14a/Nz pilot calculation
    f14a/Nz Pilot (g)
    f14a/Angle of  Attack
    f14a/Pilot G force Scope
    f14a/Pilot
    f14a/Sum1
    f14a/Stick Input
    f14a/Controller
*** 17 blocks in system 5
*** recordfile [f14a.rtw] contains 46 blocks in 5 systems
*** END LISTING
```

Congratulations! You have crafted several Target Language Compiler programs that access *model*`.rtw` files, and run them from the command line.

Inlining S-Functions with TLC

Objective: To understand the differences between a noninlined S-function and an inlined S-function. This helps you develop a better understanding of when and how to inline S-functions.

Example directory: `tlctutorial/timestwo`

In this exercise, you will generate three versions of C code for an existing S-function called `timestwo`. The first version will not use inlined code. The second version will inline the function. The third version will inline a modification that you make to the model:

- "Noninlined Code Generation" — Via simStructs and generic API
- "Why Use TLC to Implement S-functions?" — Benefits of inlining
- "Creating an Inlined S-Function" — Via custom TLC code
- "Scalar or Vector, It's the Same" — TLC adapts to the situation

You'll learn the most by performing the exercise in the order presented.

A later exercise provides information and practice in the related area of "wrapping" S-functions.

Noninlined Code Generation

1 Copy the tutorial directory `tlctutorial/timestwo` into your working directory. It contains a Simulink S-function, `timestwo.c.`

Note The `timestwo.c` code is a level-2 S-function. For general information on level-2 S-functions, see the Simulink Writing S-functions documentation.

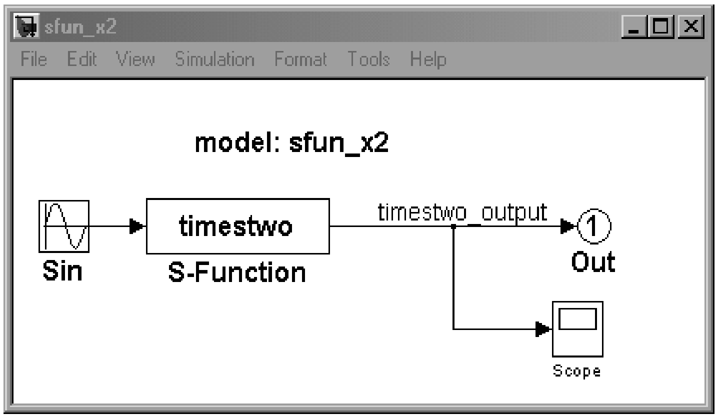
---

2 In the MATLAB command window, create a MEX-file for the S-function:

```
mex timestwo.c
```

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

3 Open the model `sfun_x2.mdl`, which uses the `timestwo` S-function. The block diagram looks like this.



4 Set up simulation parameters to use a fixed-step discrete solver with a step size of `0.01` and stop time of `10.0`.

5 Go to the **Advanced** tab of the **Simulation Parameters...** dialog, and make sure that the **Inline parameters** box is *not* selected.

6 Select **Generate code only** from the **Real-Time Workshop** tab in the **Simulation Parameters...** dialog box. The text of the **Build** button will change to **Generate code**. Click on **Generate code** to generate C source for the model. In your editor, view the `MdlOutputs` and `MdlTerminate` portions of the resulting file, `sfun_x2.c`, shown below.

Note In this and other code examples to follow, lines in bold type highlight statements that change significantly as you work the examples.

```
...
/* Compute block outputs */
void MdlOutputs(int_T tid)
{
  /* This is a single rate system */
  (void)tid;
```

```
    /* Sin: '<Root>/Sine Wave' */
    rtB.Sine_Wave = rtP.Sine_Wave_Amp *
      sin(rtP.Sine_Wave_Freq * ssGetT(rtS) + rtP.Sine_Wave_Phase) +
      rtP.Sine_Wave_Bias;

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

    /* Outport: '<Root>/Out1' */
    rtY.Out1 = rtB.S_Function;
  }

  /* Perform model update */
  void MdlUpdate(int_T tid)
  {
    /* (no update code required) */
  }

  /* Terminate function */
  void MdlTerminate(void)
  {
    /* Level2 S-Function Block: <Root>/S-Function (timestwo) */
    {
      SimStruct *rts = ssGetSFunction(rtS, 0);
      sfcnTerminate(rts);
    }
  }
```

Notice the overhead of calling the function that the S-function API necessitates (highlighted lines).

Why Use TLC to Implement S-functions?

When Real-Time Workshop generates code for a model, built-in Simulink blocks are handled differently than user-written blocks (S-functions). Each Simulink block has a corresponding *block*.tlc file that directs TLC how to encode it in a given target language. Such directives do not exist for user-written S-functions.

RTW provides a generic way to invoke user-written algorithms and drivers. Its generic API includes a variety of callback functions — for initialization, output, derivatives, termination, etc. — as well as data structures. Once coded, these are instantiated in memory and invoked during execution via indirect function calls. Each invocation involves stack frames and other overhead that adds to execution time.

In a real-time environment, especially when many solution steps are involved, generic API calls can be unacceptably slow. Real-Time Workshop can speed up S-functions in stand-alone applications it generates by embedding user-written algorithms within auto-generated functions rather than indirectly calling S-functions via the generic API. This form of optimization is called *inlining*.

You should understand that TLC is *not* a substitute for writing C code S-functions. To invoke custom blocks within Simulink, it is still necessary to code S-functions in C (or as M-files), since Simulink does not make use of TLC files. You can, however, prepare TLC files that inline specified S-functions to make your Real-Time Workshop target code much more efficient.

In the next step, you inline your S-function by removing the SimStruct associated with your S-function (achieving memory savings) and the generic API (speeding up execution).

Creating an Inlined S-Function

The Target Language Compiler creates an *inlined S-function* whenever it detects a `.tlc` file with the same name as an S-function. Assuming the `.tlc` file is properly formed, it directs construction of code that functionally duplicates the external S-function without incurring API overhead.

See how this process works by completing the following steps:

1 Find the file in your working directory called `rename_timestwo.tlc`. Rename this file to be `timestwo.tlc`, so that it will be used when generating code. The executable portion of the file is

```
%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
```

The highlighted TLC code expands to implement the algorithm.

Now create the inline version of the S-function.

2 Select **Inline parameters** from the **Advanced** tab of the **Simulation Parameters** dialog box.

3 Change the diagram's label from `model: sfun_x2` to `model: sfun_x2_ilp`.

4 Save the model as `sfun_x2_ilp.mdl`.

5 Click **Generate code**. Source files are created in a new subdirectory called `sfun_x2_ilp_grt_rtw`. Inspect the generated file `sfun_x2_ilp.c`, paying attention to the highlighted lines corresponding to those above:

```
...
/* Compute block outputs */
void MdlOutputs(int_T tid)
{
  /* This is a single rate system */
  (void)tid;
  /* Sin: '<Root>/Sin' */
  rtB.Sin = 1.0 *
    sin(1.0 * ssGetT(rtS) + 0.0) + 0.0;

  /* S-Function Block: <Root>/S-Function */
  /* Multiply input by two */
  rtB.timestwo_output = rtB.Sin * 2.0;
```

```
    /* Outport: '<Root>/Out' */
    rtY.Out = rtB.timestwo_output;
  }
  /* Perform model update */
  void MdlUpdate(int_T tid)
  {
    /* (no update code required) */
  }

  /* Terminate function */
  void MdlTerminate(void)
  {
    /* (no terminate code required) */
  }
```

Note When Real-Time Workshop generates code and builds executables, it creates or uses a specific subdirectory (called the *build directory*) to hold source, object, and make files. By default, the build directory is named *`model`*`_grt_rtw`. To find code generated by this example, look in the subdirectories `sfun_x2_grt_rtw` and `sfun_x2_ilp_grt_rtw`.

Continue the exercise by creating a stand-alone simulation, as follows.

6 Clear **Generate code only** from the Real-Time Workshop pane in the **Simulation Parameters...** dialog.

7 Go to **Workspace I/O** tab and select the **Output** box under **Save to workspace** (this causes the model's output data to be logged in your MATLAB workspace).

8 Click **Build** to generate source, compile, and link the model into an executable, named `sfun_x2_ilp.exe` (or, on UNIX systems, `sfun_x2_ilp`).

9 Confirm that the `timestwo.tlc` file did the right thing by running the stand-alone executable. To run it, in the MATLAB command window, type:

```
!sfun_x2_ilp
```

The following responses appear:

```
** starting the model **
** created sfun_x2_ilp.mat **
```

View or plot the contents of the `sfun_x2_ilp.mat` file to verify that the stand-alone model generated sine output ranging from -2 to +2. In the MATLAB command window, type

```
load sfun_x2_ilp.mat
plot (rt_yout)
```

Scalar or Vector, It's the Same

Suppose your model's input signal was a vector instead of a scalar. How would TLC handle it when producing inline code? Explore how signal vectorization works through the following exercise:

1. Open the model `sfun_x2v.mdl` that is in your working directory. This model is just a vectored version of `sfun_x2.mdl`, in which the sinewave source produces a vector of signals of width 4. It does not use inline parameters.
2. Activate the Scope block and run the simulation. Note that vector output is produced.
3. Build a stand-alone version of `sfun_x2v.mdl`. Run the executable that is produced and view its output in the MATLAB workspace.
4. Inspect the source code TLC produces (`sfun_x2v.c`), comparing it to `sfun_x2.c`. Note how the TLC implemented vector input without any changes having to be made to `timestwo.tlc`.
5. Close both models (`sfun_x2` and `sfun_x2v`) before continuing with the tutorial.

Exploring Variable Names and Loop Rolling

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

Example directory: `tlctutorial/mygain`

You will work with the model `tlctutorial/mygain/simple_sin.mdl`. It has one source (a sine wave generator), a gain block, and an output port.

The exercise guides you through following steps:

- "Getting Started" — Setting up the exercise and running the model
- "Modify the Model" — Changing the input width and seeing consequences
- "More on TLC Loop Rolling" — Parameterizing loop behavior

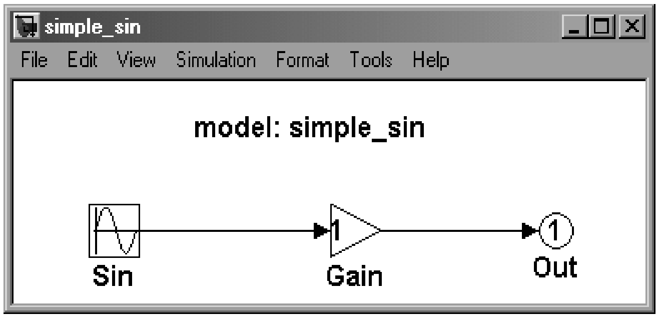
Getting Started

1 Copy the entire directory `tlctutorial/mygain` to your working directory, so that you can use the files provided.

Note You must use or create a working directory outside of *`matlabroot`*`/` for models you make. Simulink complains and refuses to build models in its own source directories

---

2 Open the model file `simple_sin.mdl`, which consists of three blocks.



3 Choose the **Target configuration** category from the **Real-Time Workshop** tab of the **Simulation parameters** dialog box. Select **Generate code only**.

4 Generate code for the model and inspect the `MdlOutputs()` section of output file `simple_sin.c` (in your build directory).

```
/* Compute block outputs */
void MdlOutputs(int_T tid)
{
  /* local block i/o variables */
  real_T rtb_Sin;

  /* This is a single rate system */
  (void)tid;

  /* Sin: '<Root>/Sin' */
  rtb_Sin = rtP.Sin_Amp *
    sin(rtP.Sin_Freq * ssGetT(rtS)+rtP.Sin_Phase)+rtP.Sin_Bias;

  /* Outport: '<Root>/Out' incorporates:
   *   Gain: '<Root>/Gain'
   *
   * Regarding '<Root>/Gain':
   *   Gain value: rtP.Gain_Gain
   */
  rtY.Out = (rtP.Gain_Gain * rtb_Sin);
}
```

Note that:

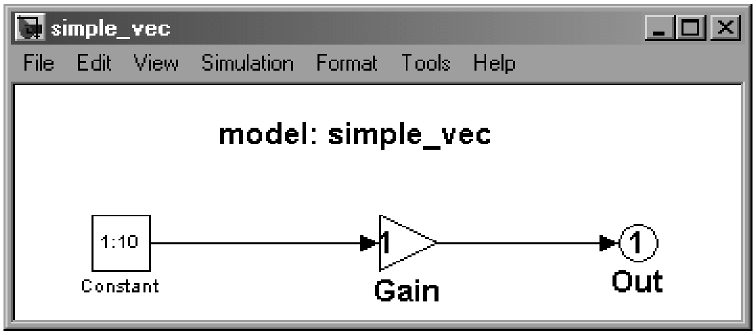
- Comments are generated, because the **Generate comments** Real-Time Workshop code generation option is on by default.
- No loops are generated (because the input and output signals are scalar).

Modify the Model

1 Delete the Sine Wave block, then replace it with a Constant block from the Library Browser.

2 Set the parameter for the constant block to `1:10`, and change the top label, `model: simple_sin`, to `model: simple_vec`.

3 Rename the model; save your changes to `simple_vec.mdl` (in your working directory). The diagram now looks like this.



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

```
/* Compute block outputs */
void MdlOutputs(int_T tid)
{
  /* This is a single rate system */
  (void)tid;

  /* Outport: '<Root>/Out' incorporates:
   *   Gain: '<Root>/Gain'
   *   Constant: '<Root>/Constant'
   *
   * Regarding '<Root>/Gain':
   *   Gain value: rtP.Gain_Gain
   */
  {
    int_T i1;
    real_T *y0 = &rtY.Out[0];
```

```
        for (i1=0; i1 < 10; i1++) {
          y0[i1] = (rtP.Gain_Gain * rtP.Constant_Value[i1]);
        }
      }
    }
```

Notice that:

- By default (when the number of iterations exceeds 5), the code that generates model outputs gets “rolled” into a loop.
- Loop index `i1` runs from 0 to 9.
- Pointer `*y0` is used and initialized to the output signal array.

Change the Loop Rolling Threshold

Real-Time Workshop either iterates or loops depending on the current **RollThreshold** value. You can change looping behavior by setting your own **RollThreshold**, which is applied to the entire model.

Note It is not possible to modify `RollThreshold` for specific blocks from the GUI

Do this for your model as follows:

1 Select the **General code generation options** category from the **Real-Time Workshop** tab of the **Simulation parameters...** dialog. Set **Loop rolling threshold** to `12`.

   Now, loops will be generated only when the width of signals passing through a block exceeds 12.

2 Click on **Generate Code** once more to recreate the output.

3 Inspect `simple_vec.c` in your editor. It will look like this:

```
/* Compute block outputs */
void MdlOutputs(int_T tid)
{
  /* This is a single rate system */
  (void)tid;

  /* Outport: '<Root>/Out' incorporates:
   *   Gain: '<Root>/Gain'
   *   Constant: '<Root>/Constant'
   *
   * Regarding '<Root>/Gain':
   *   Gain value: rtP.Gain_Gain
   */
  rtY.Out[0] = (rtP.Gain_Gain * rtP.Constant_Value[0]);
  rtY.Out[1] = (rtP.Gain_Gain * rtP.Constant_Value[1]);
  rtY.Out[2] = (rtP.Gain_Gain * rtP.Constant_Value[2]);
  rtY.Out[3] = (rtP.Gain_Gain * rtP.Constant_Value[3]);
  rtY.Out[4] = (rtP.Gain_Gain * rtP.Constant_Value[4]);
  rtY.Out[5] = (rtP.Gain_Gain * rtP.Constant_Value[5]);
  rtY.Out[6] = (rtP.Gain_Gain * rtP.Constant_Value[6]);
  rtY.Out[7] = (rtP.Gain_Gain * rtP.Constant_Value[7]);
  rtY.Out[8] = (rtP.Gain_Gain * rtP.Constant_Value[8]);
  rtY.Out[9] = (rtP.Gain_Gain * rtP.Constant_Value[9]);
}
```

4 To activate loop rolling again, change the **RollThreshold** to `10` (or less) in the **General code generation options** dialog box.

5 Modify the model to use vectored gain by double-clicking the Gain block and typing `1:10` in the **Gain** parameter field.

6 Generate the code again, and inspect it:

```
/* Compute block outputs */
void MdlOutputs(int_T tid)
{
  /* This is a single rate system */
  (void)tid;

  /* Outport: '<Root>/Out' incorporates:
   *   Gain: '<Root>/Gain'
   *   Constant: '<Root>/Constant'
   *
   * Regarding '<Root>/Gain':
   *   Gain value: 1:10
```

```
     */
    {
      int_T i1;
      real_T *y0 = &rtY.Out[0];

      for (i1=0; i1 < 10; i1++) {
        y0[i1] = (rtP.Gain_Gain[i1] * rtP.Constant_Value[i1]);
      }
    }
  }
```

Note that now there are 10 gain values, placed in the `rtP` parameter array.

7. The Gain block TLC library file *`matlabroot`*`/rtw/c/tlc/blocks/gain.tlc` has three `rollVars`: `U`, `Y`, and `P`. This indicates that inputs (`U`), parameters (`P`), and outputs (`Y`) can all be rolled when the number of signals operated on by this block exceeds the `RollThreshold`, as is true for most block functions.

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

More on TLC Loop Rolling

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

```
%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 %% Outputs
```

Arguments for %roll

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

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

1. TLC uses the first argument, `sigIdx`, to specify the appropriate index into a (signal) vector that is used in the generated code. If the signal is scalar, when analyzing that block of the *`model`*`.rtw` file, the TLC determines that only a single line of code is required. In this case, it sets `sigIdx` to `0` so as to access only the first element of a vector, and no loop is constructed.

2. The second argument of `%roll`, `lcv`, is generally specified in the `%roll` directive as `lcv = RollThreshold`. `RollThreshold` is global (model-wide) threshold with the default value of `5`. Therefore, whenever a block contains more than five contiguous and rollable variables, TLC collapses the lines nested between `%roll` and `%endroll` into a loop. If fewer than five contiguous rollable variables exist, `%roll` does *not* create a loop and instead produces individual lines of code.

3. The third argument should just be `block`. This tells TLC that it is operating on block objects. TLC code for S-functions simply use this argument as shown.

4. The fourth argument of `%roll` is a string, "`Roller`". This, specified in `rtw/c/tlc/roller.tlc`, formats the loop. Normally you pass this as is, but other loop control constructs are possible for advanced uses (see `LibBlockInputSignal` in the "Target Language Compiler Functions" on page 9-4.

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

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

   This list tells TLC that it is rolling through input signals (`Us`) and output signals (`Ys`). In cases where blocks specify an array of parameters instead of a scalar parameter, `rollvars` is specified as:

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

Input Signals, Output Signals, and Parameters

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

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

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

1. The first argument 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.

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

3. The third argument to the functions is `lcv` (loop control variable). As described previously, `lcv = RollThreshold` was set in `%roll` to indicate that a loop be constructed whenever `RollThreshold` (default value of `5`) is exceeded. For example, if there are six contiguous inputs into the block, they are rolled into a loop.

4. The fourth argument, `sigIdx`, enables the TLC to handle special cases. In the event that the `RollThreshold` is *not* exceeded (for example, if the block is only connected to a scalar input signal) the TLC does not roll it into a loop. Instead, the TLC provides an integer value for the appropriate index variable in a corresponding line of “inline” code. Whenever the `RollThreshold` is exceeded, the TLC creates a `for`-loop and uses an index variable to access inputs, outputs and parameters within the loop.

For further details, see “Target Language Compiler Functions” on page 9-4.

Debugging Your TLC Code

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

Example directory: `tlctutorial/tlcdebug`

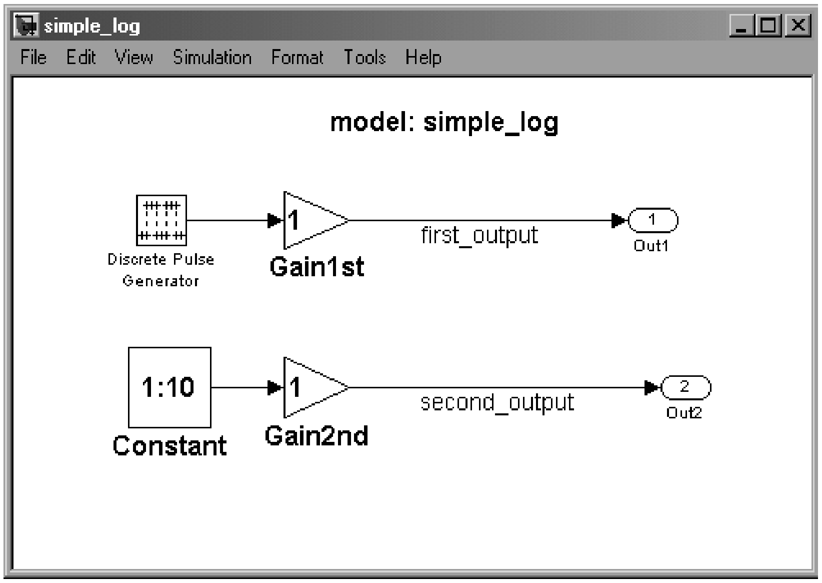
You can cause the TLC debugger to be invoked whenever Real-time Workshop builds a model. In this exercise you use it to detect a bug in a .tlc file for a model called `simple_log`. The bug causes output from the stand-alone version of the model that Real-time Workshop generates to differ from its output in Simulink:

- "Getting Started" — Run the model and inspect output
- "Generate and Run Code from the Model" — Compare compiled results to original output
- "Start the Debugger and Explore Commands" — Things you can do with the debugger
- "Debug gain.tlc" — Find out what went wrong
- "Fix the Bug and Verify the Fix" — Easy ways to fix bugs and verify fixes

Getting Started

1 If you haven't done so already, first copy the folder `tlctutorial/tlcdebug/` to your working directory and `cd` to it.

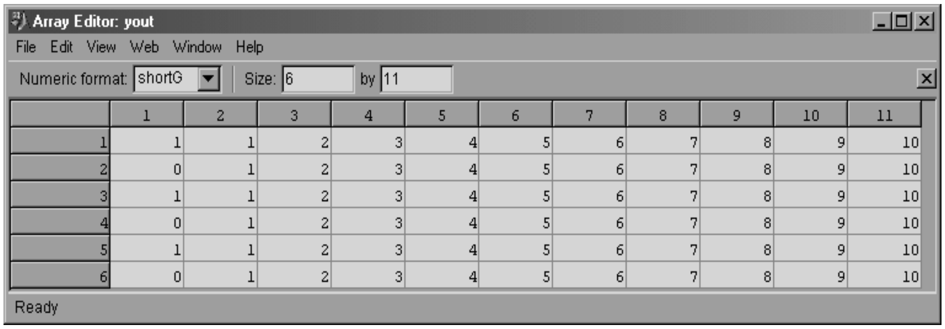
2 Open the model `simple_log.mdl` in Simulink. The diagram looks like this.



3 Open the **Simulation parameters** dialog box and click the **Workspace I/O** tab. Select the **Time** and **Output** boxes under **Save to workspace** if they are not already selected. This causes model variables to be logged to your MATLAB workspace.

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

5 Double-click `yout` in the **Workspace** pane of the MATLAB command window. The Array Editor displays the 6x11 array output from `simple_log`. You will see the following display.



	1	2	3	4	5	6	7	8	9	10	11
1	1	1	2	3	4	5	6	7	8	9	10
2	0	1	2	3	4	5	6	7	8	9	10
3	1	1	2	3	4	5	6	7	8	9	10
4	0	1	2	3	4	5	6	7	8	9	10
5	1	1	2	3	4	5	6	7	8	9	10
6	0	1	2	3	4	5	6	7	8	9	10

Column 1 contains discrete pulse output for six time steps (`1`s and `0`s), collected at port `out1`. Columns 2–11 contain the constant values `1:10`, collected at port `out2`.

Next, we'll generate a stand-alone version of `simple_log.mdl`. We'll execute it and compare its results to the output from Simulink displayed above.

Note For the purpose of this tutorial, the TLC file provided, `gain.tlc`, contains a bug and lacks certain functionality, so this S-function version cannot be used in actual practice. To override the standard `gain.tlc` gain block, this version must be in the same directory as the model that uses it.

Generate and Run Code from the Model

1 Make sure that `simple_log.mdl` is set up to generate code properly by opening the **Simulation parameters** dialog and clicking the **Advanced** tab. Make sure that **Inline parameters** is selected. The bug you will fix only appears when inline parameters are used.

2 Click the **Build** button.

   C Source code is generated, compiled and linked. The MATLAB command window shows the progress of the build, which ends with these messages:

```
### Created executable: simple_log.exe
### Successful completion of Real-Time Workshop build procedure
for model: simple_log
```

3 Run the stand-alone model just created by typing

```
!simple_log
```

This results in the messages

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

4 Inspect results by placing the variables in your workspace; double-click on `simple_log.mat` in your Current Directory pane, then double-click on `rt_yout` (the stand-alone version of variable `yout`) in the Workspace pane.

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

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

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

```
void MdlOutputs(int_T tid)
{
  /* local block i/o variables */
  real_T rtb_Discrete_Pulse_Generator;
  real_T rtb_first_output;

  /* This is a single rate system */
  (void)tid;

  /* DiscretePulseGenerator: '<Root>/Discrete Pulse Generator' */
  rtb_Discrete_Pulse_Generator =
    (rtDWork.Discrete_Pulse_Generator_IWORK.ClockTicksCounter < 1.0 &&
    rtDWork.Discrete_Pulse_Generator_IWORK.ClockTicksCounter >= 0) ?
   1.0 :
    0.0;
  if (rtDWork.Discrete_Pulse_Generator_IWORK.ClockTicksCounter >= 2.0-1) {
    rtDWork.Discrete_Pulse_Generator_IWORK.ClockTicksCounter = 0;
  } else {
    (rtDWork.Discrete_Pulse_Generator_IWORK.ClockTicksCounter)++;
  }
```

```
    /* Gain Block: <Root>/Gain1st */

    rtb_first_output = 1.0;

    /* Outport: '<Root>/Out1' */
    rtY.Out1 = rtb_first_output;
  }
```

Note the line near the end, `rtb_first_output = 1.0;`. How did a constant value get passed to the output when it was supposed to receive a variable that alternates between `1.0` and `0.0`? Use the debugger to find out.

Start the Debugger and Explore Commands

We use the TLC debugger to monitor the code generation process. As it is not invoked by default, we need to request the debugger to start.

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

   Click the **Real-Time Workshop** tab, and 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 describes the building process. The build stops at the `gain.tlc` file and displays the command prompt:

   ```
   TLC-DEBUG>
   ```

2 Type `help` to list the TLC debugger commands. Here are some things you can do in the debugger; try them if you like:

   - View and query various entities in the TLC scope. For example:

     ```
     TLC-DEBUG> whos CompiledModel
     TLC-DEBUG> print CompiledModel.NumSystems
     TLC-DEBUG> print TYPE(CompiledModel.NumSystems)
     ```
   - Examine the statements in your current context. For example:

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

- move to the next line of code:
  ```
  TLC-DEBUG> next
  ```
- Step into a function:
  ```
  TLC-DEBUG> step
  ```
- Assign a constant value to a variable, such as the input signal `%<u>`:
  ```
  TLC-DEBUG> assign u = 5.0
  ```
- Set a breakpoint where you are or in some other part of the code:
  ```
  TLC-DEBUG> break
  TLC-DEBUG> break gain.tlc:40
  ```
- Execute until the next breakpoint:
  ```
  TLC-DEBUG> continue
  ```
- Clear breakpoints you have established:
  ```
  TLC-DEBUG> clear 1
  TLC-DEBUG> clear all
  ```

Debug gain.tlc

Now let's look around to find out what is wrong with our code:

1 Set a breakpoint on line 49 of `gain.tlc`:

```
TLC-DEBUG> break gain.tlc:49
```

Instruct the TLC debugger to advance to your breakpoint:

```
TLC-DEBUG> continue
```

TLC processes input, reports its progress, advances to line 50 in `gain.tlc`, displays the line, and pauses:

```
### Loading TMW TLC function libraries
..
### Initial pass through model to cache user defined code
### Caching model source code
Warning:  USING LOCAL VERSION OF gain.tlc
.
Breakpoint 1
00049:    %if InlineParameters == 1
```

2 Use the `whos` command to see the variables in the current scope:

```
TLC-DEBUG> whos
Variables within: <BLOCK_LOCAL>
k                               String
rollVars                        Vector
u                               String
y                               String
block                           Resolved
system                          Resolved
```

3 Inspect the variables using the `print` command (note, names are case sensitive):

```
TLC-DEBUG> print k
1.0

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

TLC-DEBUG> print u
u0[i1]

TLC-DEBUG> print y
y0[i1]
```

4 Execute one step:

```
TLC-DEBUG> step
00051:       %<FcnEliminateUnnecessaryParams(y, u, k)>\
```

(We reached this statement because `InlineParameters` is `TRUE`.)

5 Step again, into `FcnEliminateUnnecessaryParams`:

```
TLC-DEBUG> step
00024: %if LibIsEqual(k, "0.0")
```

6 Rather than stepping into `LibIsEqual()`, advance via the `next` command:

```
TLC-DEBUG> next
00028:   %elseif LibIsEqual(k, "1.0")
```

Examine `k`'s value:

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

```
1.0
```

7 Since `k` does equal `1.0`, the statement following the `%if` is executed:

```
TLC-DEBUG> next
00029:     %<y> = %<k>;
```

Examine y's value:

```
TLC-DEBUG> print y
y0[i1]
```

8 Advance once more time and you'll see that this function is finished:

```
TLC-DEBUG> next
00051:       %<FcnEliminateUnnecessaryParams(y, u, k)>\
```

Clearly, this is the origin of the C statement responsible for the erroneous constant output, `rtb_first_output = 1.0;`.

9 Abandon the build by quitting the TLC. Type

```
TLC-DEBUG> quit
```

Fix the Bug and Verify the Fix

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

```
%function FcnEliminateUnnecessaryParams(y,u,k) Output
%if LibIsEqual(k, "0.0")
    %if ShowEliminatedStatements == 1
      /* %<y> = %<u> * %<k>; */
    %endif
  %elseif LibIsEqual(k, "1.0")
    %<y> = %<k>;
  %elseif LibIsEqual(k, "-1.0")
    %<y> = -%<k>;
  %else
    %<y> = %<u> * %<k>;
  %endif
%endfunction
```

Testing the constant parameter `k` against 0.0, 1.0, and -1.0 is fine; the problem occurs in the assignment of `y` to `k` in the latter two cases:

1. To fix these coding errors, edit `gain.tlc`, and make two substitutions:
   - `Change %<y> = %<k>; to %<y> = %<u>;`
   - `Change %<y> = -%<k>; to %<y> = -%<u>;`

   Now the input variable is copied or negated into the output variable instead of assigning a constant value to the output. Save `gain.tlc`.

2. Build the stand-alone model again. Go through the debugging procedure described above to verify that `y` is correctly assigned inside `FcnEliminateUnnecessaryParams`. Complete the build by typing `continue` at each `TLC-DEBUG>` prompt.

3. Execute the stand-alone model by typing `!simple_log`. A new version of `simple_log.mat` is created containing its output.

4. Compare the workspace variable `rt_yout` with `yout`, as you did before. All values in the first column should now correspond.

For more information about the TLC debugger, see Chapter 7, "Debugging TLC Files."

Using TLC Code Coverage to Aid Debugging

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

Example directory: `tlctutorial/tlcdebug`

Note If you haven't completed the previous exercise, "Debugging Your TLC Code" on page 3-40, go back and do so before continuing here.

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

- "Getting Started" — Why and how to analyze TLC coverage
- "Open the Model and Generate Code" — Reading a coverage log file
- "Log Without Inline Parameters" — How code generation is affected

Getting Started

"Debugging Your TLC Code" on page 3-40 notes that the bug in `gain.tlc` does not appear unless you use inline parameters. This happens because the faulty function, `FcnEliminateUnnecessaryParams()`, is designed to eliminate multiplication by unity, and thus requires parameters to be expressed as constants. When the `simple_log` model is built without inline parameters, the outputs are correct because that function is not called.

When you notice inconsistent behavior such as this, it can help to inquire into how the TLC behaved under various conditions. A good way to uncover such information is to turn on the TLC coverage analyzer when generating code. This brief exercise shows how to do this and how it can help find problems in TLC code.

Under the `TLC debugging` category of the **Real-Time Workshop** tab is a check box, **Start TLC coverage when generating code**. Selecting it results in a listing that documents how many times each line in your TLC source file was touched during code generation. The listing, `name.log` (where `name` is the filename of the TLC file being analyzed), is placed in your build directory.

Note In fact, a log file for *every* `.tlc` file invoked or included is generated in the build directory. Here we are only interested in the one called `gain.log`.

Open the Model and Generate Code

1. Copy the folder `tlctutorial/tlcdebug/` to your working directory and `cd` to it. *Do this even if you already have copied it*, to be sure you have the version of `gain.tlc` that has the bug. Open `simple_log.mdl` in Simulink.
2. As before, open the **Simulation parameters** dialog box, click the **Advanced** tab and select **Inline parameters.**
3. Under the **Target Configuration** section of the **Real-Time Workshop** tab, click **Generate code only**.
4. Under the **TLC debugging** section of the **Real-Time Workshop** tab, select **Start TLC coverage when generating code**. Do *not* check **Start TLC debugger when generating code** — it won't hurt to invoke the debugger, but you won't need it.
5. Click **Generate code**. The usual messages appear on the command window, and a build directory (`simple_log_grt_rtw`) is created in your working directory.
6. Enter the build directory. Find the file `gain.log`, and copy it to your working directory, renaming it `gain_ilp.log` to prevent it from being overwritten.
7. Open the log file `gain_ilp.log` in your editor. It looks almost like `gain.tlc`, except for a number followed by a colon at the beginning of each line. This number represents the number of times the TLC executed the line in generating code. `FcnEliminateUnnecessaryParams`: should look like this:

```
0: %% Function: FcnEliminateUnnecessaryParams =================
0: %% Abstract:
0: %%      Eliminate unnecessary multiplications for following
0: %%      gain cases when in-lining parameters:
```

```
0: %%      Zero: memset in registration routine zeroes output
0: %%      Positive One: assign output equal to input
0: %%    Negative One: assign output equal to unary minus of input
0: %%
1: %function FcnEliminateUnnecessaryParams(y,u,k) Output
2: %if LibIsEqual(k, "0.0")
0:     %if ShowEliminatedStatements == 1
0:       /* %<y> = %<u> * %<k>; */
0:     %endif
2:   %elseif LibIsEqual(k, "1.0")
2:     %<y> = %<k>;
0:   %elseif LibIsEqual(k, "-1.0")
0:     %<y> = -%<k>;
0:   %else
0:     %<y> = %<u> * %<k>;
0:   %endif
0: %endfunction
```

Notice that comments were not executed, nor were most of the TLC statements. Several statements were reached, however, which means they output to the generated C code as many times as the number prepended to those lines.

Log Without Inline Parameters

1 Open the **Simulation parameters** dialog, click the **Advanced** tab and clear **Inline parameters.**

2 Go to the **Real-Time Workshop** tab, and click **Generate code**. The TLC will run through its tasks and recreate the log and source files in your build directory.

3 Enter your build directory. Find the new version of `gain.log`, and copy it to your working directory, but call it `gain_no_ilp.log`.

4 Edit the log file `gain_no_ilp.log`, which appears as shown below:

```
0: %% Function: FcnEliminateUnnecessaryParams =================
0: %% Abstract:
0: %%      Eliminate unnecessary multiplications for following
```

```
 0: %%      gain cases when in-lining parameters:
 0: %%      Zero: memset in registration routine zeroes output
 0: %%      Positive One: assign output equal to input
 0: %%    Negative One: assign output equal to unary minus of input
 0: %%
 1: %function FcnEliminateUnnecessaryParams(y,u,k) Output
 0: %if LibIsEqual(k, "0.0")
 0:     %if ShowEliminatedStatements == 1
 0:       /* %<y> = %<u> * %<k>; */
 0:     %endif
 0:   %elseif LibIsEqual(k, "1.0")
 0:     %<y> = %<k>;
 0:   %elseif LibIsEqual(k, "-1.0")
 0:     %<y> = -%<k>;
 0:   %else
 0:     %<y> = %<u> * %<k>;
 0:   %endif
 0: %endfunction
```

Compare this listing to the prior one (gain_ilp.log). Observe that this time, no statements were executed except for the function declaration, which the TLC always caches. The differences in the log files focus your attention on what is going wrong when parameters are inlined.

In this exercise, you have seen how changing code generation options can cause a latent defect to appear in generated source code. Systematically changing options and observing resulting differences in TLC coverage enhances the speed and ease of discovering faulty code.

Wrapping User Code with TLC

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

Example directory: `tlctutorial/wrapper`

Wrapper S-functions enable you to pull in existing C functions without fully rewriting them in the context of Simulink S-functions. Each wrapper you provide is an S-function “shell” that merely calls one or more existing, external functions. Wrappers are explained and demonstrated below as follows:

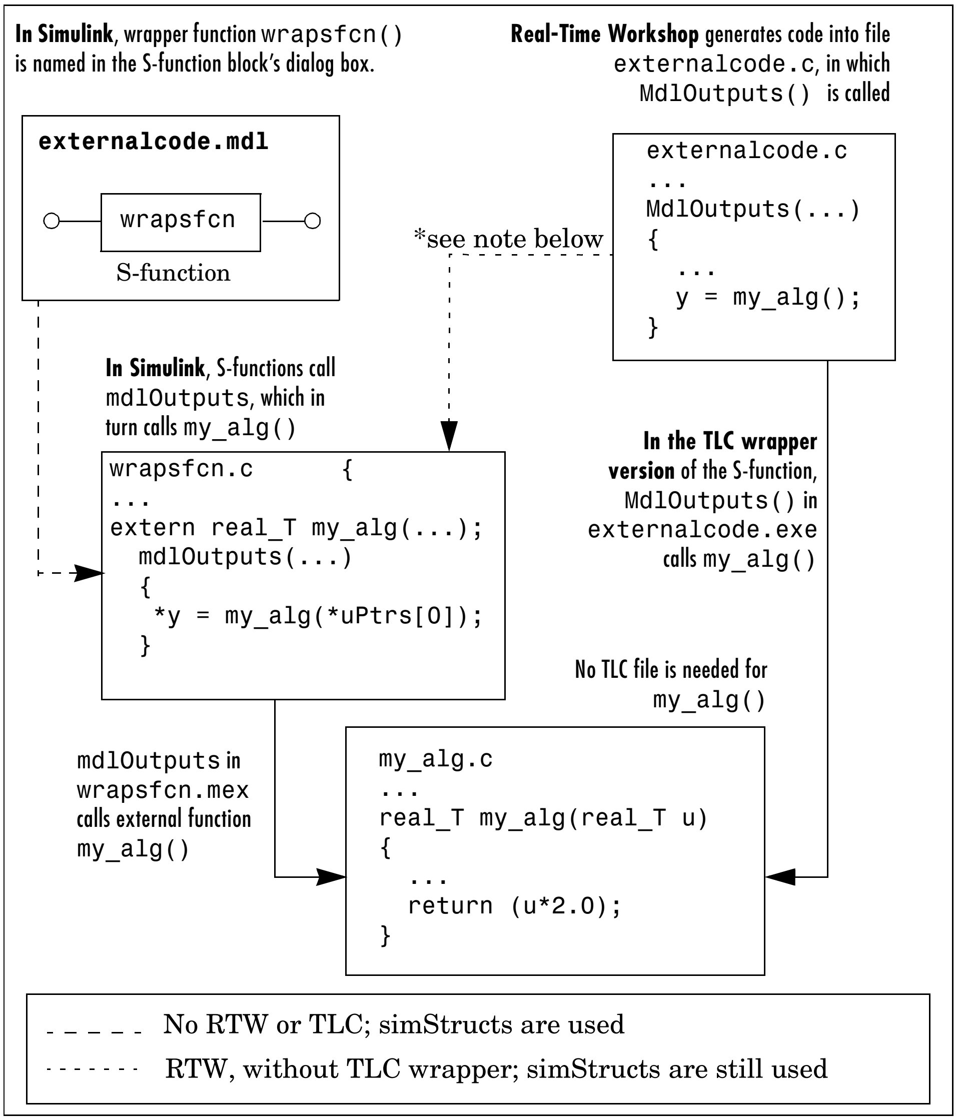
- “Why Wrap?” — Reasons for building TLC wrapper functions
- “Getting Started” — Set up the wrapper exercise
- “Generate Code Without a Wrapper” — How Real-Time Workshop handles external functions by default
- “Generate Code Using a Wrapper” — Bypassing the API overhead

Why Wrap?

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

- You can construct an S-function from user code that hooks it to the Simulink generic API. This is the simplest approach, but sacrifices efficiency when a stand-alone application is built.
- You can inline the S-function, reimplementing it as a TLC file. This improves efficiency, but takes time and effort, can introduce errors into working code, and leads to two sets of code to maintain for each algorithm.
- You can inline the S-function via a TLC *wrapper function*. By doing so, you only need to create a small amount of TLC code, and the algorithm can remain coded in its existing form.

The figure below illustrates how S-function wrappers operate.


In Simulink, wrapper function wrapsfcn() is named in the S-function block's dialog box.
Real-Time Workshop generates code into file externalcode.c, in which MdlOutputs() is called
externalcode.mdl
wrapsfcn
S-function
*see note below
In Simulink, S-functions call mdlOutputs, which in turn calls my_alg()
In the TLC wrapper version of the S-function, MdlOutputs() in externalcode.exe calls my_alg()
No TLC file is needed for my_alg()
mdlOutputs in wrapsfcn.mex calls external function my_alg()
- - - - - No RTW or TLC; simStructs are used
- - - - - - - RTW, without TLC wrapper; simStructs are still used

```
externalcode.c
...
MdlOutputs(...)
{
  ...
  y = my_alg();
}
```

```
wrapsfcn.c      {
...
extern real_T my_alg(...);
  mdlOutputs(...)
  {
   *y = my_alg(*uPtrs[0]);
  }
```

```
my_alg.c
...
real_T my_alg(real_T u)
{
  ...
  return (u*2.0);
}
```

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

For example, you might have an existing object file compiled for a TI DSP or for some other processor on which Simulink does not run. You can write a dummy C S-function and use a TLC wrapper that calls the external function, despite not having its source code. You could similarly access functions in a library of algorithms optimized for the target processor. Accomplishing this requires making appropriate changes to a template makefile, or otherwise providing a means to link against the library.

Caveat Using object files that lack source code only works at present with the Microsoft Visual C/C++ Compiler (MSVC). In this case, the object files must also have been created with the same compiler (i.e., MSVC).

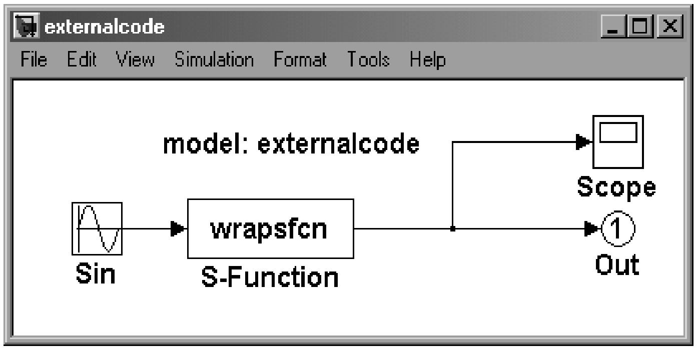
The only restriction on S-function wrappers is that they must use the correct number of block inputs and outputs. Wrapper code may include computations, but usually these are limited to transforming (e.g., scaling or reformatting) values passed to and from the wrapped external functions.

Getting Started

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

1 Copy the folder `tlctutorial/wrapper` to your working directory.

2 In MATLAB, run the model `externalcode.mdl` from your working directory. The block diagram looks like the following.



3 Run the model (from the **Simulation** menu, or type **Ctrl+T**). You will get an error telling you that `wrapsfcn` does not exist. Can you figure out why?

   Answer: `wrapsfcn` isn't available because there is no `mex` file for it.

4 To rectify this, in the MATLAB command window type

```
mex wrapsfcn.c
```

5 Run the simulation again. With the S-function present, no error occurs.

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

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

Generate Code Without a Wrapper

Before creating a wrapper, generate code that uses the Simulink generic API.

1 Build a stand-alone model (the file will be called `externalcode.exe` in Windows and `externalcode` in UNIX environments). Click **Tools -> Real-Time Workshop -> Build Model**, or type **Ctrl+B**. Simulink will report the results, as follows:

Generates Code

```
### Starting Real-Time Workshop build procedure for model: externalcode
### Generating code into build directory: .\externalcode_grt_rtw
### Invoking Target Language Compiler on externalcode.rtw
tlc -r .\externalcode_grt_rtw\externalcode.rtw
'D:\MATLAB6p1\rtw\c\grt\grt.tlc' -O.\externalcode_grt_rtw
'-ID:\MATLAB6p1\rtw\c\grt' '-ID:\MATLAB6p1\rtw\c\tlc'
-aEnforceIntegerDowncast=1 -aExtMode=0 -aFoldNonRolledExpr=1
-aForceParamTrailComments=0 -aGenerateComments=1
-aIgnoreCustomStorageClasses=1 -aIncHierarchyInIds=0
-aInlineInvariantSignals=1 -aInlineParameters=0 -aLocalBlockOutputs=1
-aLogVarNameModifier="rt_" -aRTWVerbose=1 -aRollThreshold=5
-aShowEliminatedStatements=1 -aReleaseVersion=12.1 -p10000
### Loading TMW TLC function libraries
..
### Initial pass through model to cache user defined code
### Caching model source code
.
### Creating (RealTime) source file externalcode.c
.
### Creating model header file externalcode.h
### Creating model header file externalcode_export.h
### Creating parameter file externalcode_prm.h
### Creating registration file externalcode_reg.h
### TLC code generation complete.
```

Compiles S-function and wrapper

```
### Creating project marker file: rtw_proj.tmw
### Creating externalcode.mk from D:\MATLAB6p1\rtw\c\grt\grt_lcc.tmf
### Building externalcode: .\externalcode.bat

D:\Work\wrapper\externalcode_grt_rtw>set MATLAB=D:\MATLAB6p1

D:\Work\wrapper\externalcode_grt_rtw>"D:\MATLAB6p1\rtw\bin\win32\gmake" -f
externalcode.mk
D:\MATLAB6p1\sys\lcc\bin\lcc -c -Foexternalcode.obj     -DMODEL=externalcode
-DRT -DNUMST=2 -DTID01EQ=1 -DNCSTATES=0 -DMT=0 -DHAVESTDIO -I. -I..
-ID:\MATLAB6p1\simulink\include -ID:\MATLAB6p1\extern\include
-ID:\MATLAB6p1\rtw\c\src -ID:\MATLAB6p1\rtw\c\libsrc
-ID:\MATLAB6p1\sys\lcc\include  -noregistrylookup externalcode.c
Warning:  externalcode.c: 41  Statement has no effect
0 errors, 1 warnings
```

Compiles Support files

```
D:\MATLAB6p1\sys\lcc\bin\lcc -c -Fogrt_main.obj    -DMODEL=externalcode -DRT
-DNUMST=2 -DTID01EQ=1 -DNCSTATES=0 -DMT=0 -DHAVESTDIO -I. -I..
-ID:\MATLAB6p1\simulink\include -ID:\MATLAB6p1\extern\include
-ID:\MATLAB6p1\rtw\c\src -ID:\MATLAB6p1\rtw\c\libsrc
-ID:\MATLAB6p1\sys\lcc\include -noregistrylookup
D:\MATLAB6p1/rtw/c/grt/grt_main.c
D:\MATLAB6p1\sys\lcc\bin\lcc -c -Fort_sim.obj     -DMODEL=externalcode -DRT
-DNUMST=2 -DTID01EQ=1 -DNCSTATES=0 -DMT=0 -DHAVESTDIO -I. -I..
-ID:\MATLAB6p1\simulink\include -ID:\MATLAB6p1\extern\include
-ID:\MATLAB6p1\rtw\c\src -ID:\MATLAB6p1\rtw\c\libsrc
-ID:\MATLAB6p1\sys\lcc\include  -noregistrylookup
D:\MATLAB6p1/rtw/c/src/rt_sim.c
D:\MATLAB6p1\sys\lcc\bin\lcc -c -Fortwlog.obj     -DMODEL=externalcode -DRT
-DNUMST=2 -DTID01EQ=1 -DNCSTATES=0 -DMT=0 -DHAVESTDIO -I. -I..
-ID:\MATLAB6p1\simulink\include -ID:\MATLAB6p1\extern\include
-ID:\MATLAB6p1\rtw\c\src -ID:\MATLAB6p1\rtw\c\libsrc
-ID:\MATLAB6p1\sys\lcc\include  -noregistrylookup
D:\MATLAB6p1/rtw/c/src/rtwlog.c
D:\MATLAB6p1\sys\lcc\bin\lcc -c -Fort_nonfinite.obj    -DMODEL=externalcode
-DRT -DNUMST=2 -DTID01EQ=1 -DNCSTATES=0 -DMT=0 -DHAVESTDIO -I. -I..
-ID:\MATLAB6p1\simulink\include -ID:\MATLAB6p1\extern\include
-ID:\MATLAB6p1\rtw\c\src -ID:\MATLAB6p1\rtw\c\libsrc
-ID:\MATLAB6p1\sys\lcc\include  -noregistrylookup
D:\MATLAB6p1/rtw/c/src/rt_nonfinite.c
```

Links

```
D:\MATLAB6p1\sys\lcc\bin\lcclnk -s -LD:\MATLAB6p1\sys\lcc\lib -o
../externalcode.exe externalcode.obj  grt_main.obj rt_sim.obj rtwlog.obj
rt_nonfinite.obj      D:\MATLAB6p1\rtw\c\lib\win32\rtwlib_lcc.lib
### Created executable: externalcode.exe
### Successful completion of Real-Time Workshop build procedure for model:
externalcode
```

Real-Time Workshop creates the stand-alone program, `externalcode.exe`, in your working directory and places the source and object files in your build directory.

2 Run the stand-alone to see that it behaves the same as the Simulink version. There should be no differences:

```
!externalcode

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

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

```
static void mdlOutputs(SimStruct *S, int tid)
{
    int_T             i;    /* not needed */
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
    real_T            *y    = ssGetOutputPortRealSignal(S,0);
    int_T             width = ssGetOutputPortWidth(S,0);

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

Notice this line in `wrapsfcn.c`:

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

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

```
/*
 * Copyright 1994-2001 The MathWorks, Inc.
 * $Revision: 1.3 $
 */

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

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

Generate Code Using a Wrapper

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

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

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

1. Open the file `change_wrapsfcn.tlc` in your editor, and add lines of code where comments indicate to create a workable wrapper.
2. Save the edited file as `wrapsfcn.tlc`. It must have the same name as the S-function block that uses it or the TLC won't be called to inline code.
3. Inform Simulink that your code has an external reference to be resolved. To update the model's parameters, in the MATLAB command window type

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

4. Click **Build Model** (or type **Ctrl+B**) to create the stand-alone application.
5. If all went well, run the new stand-alone application and verify that it yields identical results as prior executions:

```
!externalcode
```

If you had any problems building the application:

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

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

```
%% File    : wrapsfcn.tlc
%% Abstract:
%%      Example tlc file for S-function wrapsfcn.c
%%
%% Copyright 1994-2001 The MathWorks, Inc.
%%
%% $Revision: 1.3 $

%implements "wrapsfcn" "C"

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

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


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

%endfunction %% Outputs
```

Look at the highlighted lines. Did you declare `my_alg()` as `extern double`? Did you call `my_alg()` with the proper input and output? Correct any mistakes you may have made and build the model again.

Code Generation Architecture

The following sections provide a high-level view of how TLC orchestrates automatic program building, describing how it is invoked, what files are involved, and how it transforms records in `model.rtw` files.

Build Process (p. 4-2)	How the Target Language Compiler processes compiled model files to produce code
Invoking Code Generation (p. 4-9)	Running `rtwgen` and `tlc` from the MATLAB command line
Configuring TLC (p. 4-11)	How to pass in configuration data to customize builds
Code Generation Concepts (p. 4-13)	Understanding TLC variables and file and record handling
TLC Files (p. 4-19)	The roles and varieties of system and block target files
Data Handling with TLC: An Example (p. 4-27)	One way TLC library functions can transform data into data structures

Build Process

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

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

The Target Language Compiler is a separate binary program that is included as a MEX-file. The Compiler compiles files written in the target language. The target language is an interpreted language, and thus, the Compiler operates on source files every time it executes. You can make changes to a target file and watch the effects of your change the next time you build a model. You do not need to recompile the Target Language Compiler binary or any other such large binary to see the effects of your change.

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

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

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

A Basic Example

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

Process

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

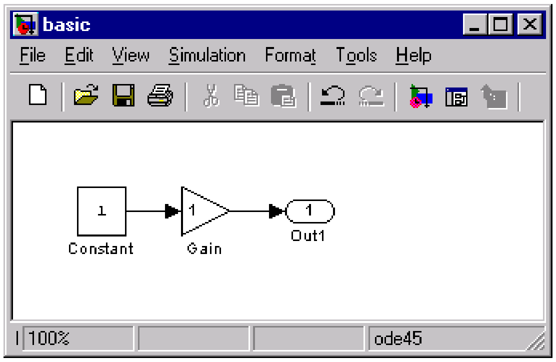
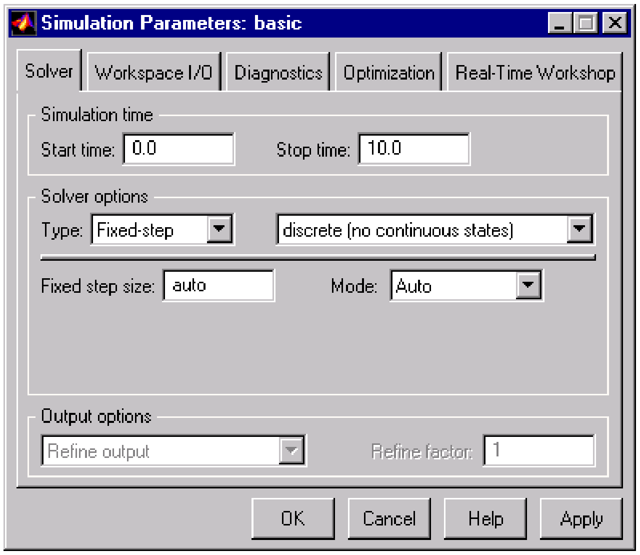


Figure 4-1: Simulink Model

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



Simulation time
Start time: 0.0 Stop time: 10.0

Solver options
Type: Fixed-step discrete (no continuous states)
Fixed step size: auto Mode: Auto

Output options
Refine output Refine factor: 1

Figure 4-2: Simulation Parameters Dialog Box

Select `Fixed-step` from the **Solver** pane of the **Simulation Parameters** dialog box and then click the **Real-Time Workshop** tab. Then, click the **Generate**

Code only button. From the **Category** drop down list select the `TLC Debugging` option. The dialog changes, allowing you to select the **Retain .rtw file** option. After doing so, 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 displays:

```
### 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 hierarchy of records it contains includes among others, the following elements (where elided lines are denoted by "..."; comments are delimited by < > and do not appear in the file).

```
CompiledModel {
  <general model information, such as>
  Name          "basic"
  Version  "5.0 $Date: 2001/12/05 22:07:01 $"
  ModelVersion  "1.1"
  GeneratedOn  "Thu Feb 07 09:17:27 2002"
  ExprFolding  1
  TargetStyle  StandAloneTarget
  Solver   FixedStepDiscrete
  SolverType  FixedStep
  StartTime  0.0
  StopTime  10.0
  ...
  DataTypes {
    <definitions of datatypes used>
    NumDataTypes    13
    NumSLBuiltInDataTypes   9
    StrictBooleanCheckEnabled 0
    DataType {
      DTName      double
      Id       0
      IdAliasedTo      -10
      IdResolvesTo      0
      IsSigned      0
      RequiredBits      0
      FixedExp      0
      FracSlope      1
      Bias      0
      IsFixedPoint      0
    }
    ...
  }
```

```
<Global properties of blocks, targets, inputs, outputs>
...
BlockOutputs {
  <Specific block outputs>
  ...
}
DWorks {
  <DWorks storage description>
  ...
}
<States and Events>
ContStates {
  NumContStates    0
}
NonsampledZCs {
  NumNonsampledZCs    0
}
ZCEvents {
  NumZCEvents    0
}
BlockDefaults {
  <Defaults for blocks>
  ...
}
ParameterDefaults {
      <Defaults for parameters>
  ...
}
DataInputPortDefaults {
      <Defaults for input ports>
  ...
}
<Other defaults>
...
ModelParameters {
  NumParameters    2
  ...
  ParameterDefaults {
    ...
  }
  <Actual parameters>
  Parameter {
    ...
  }
  Parameter {
    ...
  }
}
<Signals>
...
<Subsystems>
...
<Subsystem blocks>
```

```
NumSystems  1
System {
  Type     root
  Name     "<Root>"
  ...
  NumTotalBlocks    3
  Block {
    Type      Constant
    BlockIdx      [0, 0, 0]
    ...
    Parameter {
      Name       "Value"
       Value           [1.0]
      String     "1"
      StringType"Expression"
      NeedParenthesis0
    }
    Value      Parameter[0]
  }
  Block {
    Type      Gain
    BlockIdx      [0, 0, 1]
    ...
    Name      "<Root>/Gain"
    Identifier      Gain
    TID      0
    RollRegions      [0]
    NumDataInputPorts      1
    ...
    ParamSettings {
      Multiplication"Element-wise(K.*u)"
      SaturateOnOverflowNotNeeded
    }
    Parameters      [1, 1]
    Parameter {
      Name       "Gain"
      ...
      Value      [1.0]
      String     "1"
      StringType"Expression"
      NeedParenthesis0
    }
    Gain      Parameter[0]
  }
  Block {
    Type      Outport
    BlockIdx      [0, 0, 2]
    ...
    Name      "<Root>/Out1"
    Identifier      Out1
    TID      0
    RollRegions      [0]
    NumDataInputPorts      1
```

```
          ...
          ParamSettings {
            VirtualizableRootno
            PortNumber1
            OutputLocationY0
            SpecifyICno
          }
        }
        EmptySubsysInfo {
          NumRTWdatas      0
        }
      }
      BlockParamChecksum  Vector(4)
    ["1563106944U", "312355339U", "1175093170U", "625369615U"]
      ModelChecksum  Vector(4)
    ["2667851175U", "2830856923U", "1096099938U", "890634863U"]
    }
```

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. Note that a space must separate `-r` and the input filename. The `-v` option tells TLC to be verbose in reporting its activity.

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

```
My model is called basic.
It was generated on Mon Dec 03 09:42:13 2001.

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

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

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

Invoking Code Generation

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

The rtwgen Command

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

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

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

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

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

```
help rtwgen
```

at the MATLAB prompt.

The tlc Command

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

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

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

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

- `-aident=expression`, which assigns a value to the TLC identifier `ident`. Note that there is *no* space after `-a`. This is discussed in “Configuring TLC” on page 4-11 and explored in the TLC tutorial in Chapter 3, “Processing model.rtw Files with TLC Scripts.”

For more details, type

```
help tlc
```

at the MATLAB prompt.

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 7, "Debugging TLC Files."

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

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

```
%if !EXISTS(myConfigVariable)
  %assign myConfigVariable = 0
%endif

%if (myConfigVariable == 1)

    code fragment 1

%else

    code fragment 2

%endif
```

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

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

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

Code Generation Concepts

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

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

Output Streams

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

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

To run this Target Language Compiler program, type

```
tlc hello.tlc
```

at the MATLAB prompt.

This simple script demonstrates some important concepts underlying the purpose (and hence the design) of the Target Language Compiler. Since the primary purpose of the Target Language Compiler is to generate code, it is output (or stream) oriented. It makes it easy to handle buffers of text and output them easily. In the above script, the `%selectfile` directive tells the Target Language Compiler to send any following text that it generates or does not recognize to the standard output device. All syntax that the Target Language Compiler recognizes begins with the `%` character. Since `Hello, World` is not recognized, it is sent directly to the output. You could just as easily change the output destination to be a file. The `STDOUT` stream does not have to be opened, but must be selected in order to write to the command window.

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

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

Variable Types

The absence of explicit type declarations for variables is another feature of the Target Language Compiler. See Chapter 6, "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. Record processing is summarized and illustrated in the tutorial “Reading Record Files with TLC” on page 3-3 and in “Processing model.rtw Files with TLC Scripts” on page 3-18.

Records can have nested records, or subrecords, within them. The *`model`*`.rtw` file is essentially one large record, named `CompiledModel`, containing levels of subrecords. Thus, a simple script that loops through a model and outputs the name of all blocks in the model would have the following form. The tutorial “Processing model.rtw Files with TLC Scripts” on page 3-18 presents and discusses in detail the working of such a script.

```
%include "utillib.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, `utillib.tlc` contains the definition of `LibGetFormattedBlockPath`.

Like Pascal, the Target Language Compiler provides a `%with` directive that facilitates using records. See Chapter 6, “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 current structure of the *`model`*`.rtw` file including all the field names and the interpretation of their values. The format and structure of this file are subject to change from one release of Real-Time Workshop to another.

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

Functions such as `LibGetFormattedBlockPath` are provided in the Target Language Compiler libraries located in *`matlabroot`*`/rtw/c/tlc/lib/*.tlc`. For a complete list of available functions, refer to 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 expression references a variable in one of the following forms:

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

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

- A qualified variable followed by a bracketed expression such as

  ```
  var[expr]
  ```

Record Aliases

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

The following code fragment illustrates the use of aliases.

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

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

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

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

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

It is possible to create aliases to records which are not attached to any other records, as in the following example.

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

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

Saving this script as alias_func.tlc and invoking it with

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

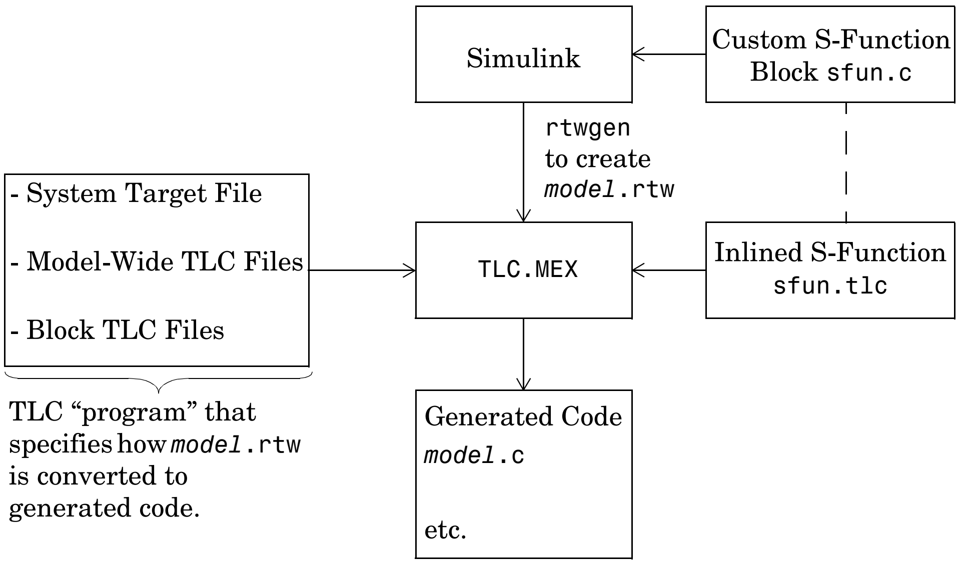
produces the command window output

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

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

TLC Files

The Target Language Compiler works with Simulink to generate code as shown in the following figure.


Simulink
Custom S-Function Block sfun.c
rtwgen to create model.rtw
- System Target File
- Model-Wide TLC Files
- Block TLC Files
TLC.MEX
Inlined S-Function sfun.tlc
TLC "program" that specifies how model.rtw is converted to generated code.
Generated Code model.c etc.

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

Available Target Files

Target files are the set of files that are interpreted by the Target Language Compiler to transform the intermediate Real-Time Workshop code (`model.rtw`) produced by Simulink into target-specific code.

Target files provide you with the flexibility to customize the code generated by the Compiler to suit your specific needs. By modifying the target files included with the Compiler, you can dictate what the compiler produces. For example, if you use the available system target files, you produce generic C code from your Simulink model. This executable C code is not platform specific.

All of the parameters used in the target files are read from the *`model`*`.rtw` file and looked up using block scoping rules. You can define additional parameters within the target files using the `%assign` statement. The block scoping rules and the `%assign` statement are discussed in Directives and Built-In Functions.

Target files are written using target language directives. Chapter 6, "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 Target Language Compiler. It is analogous to the `main()` routine of a C program. System target files oversee the entire code generation process. For example, the system target file, `grt.tlc`, sets up some variables for `codegenentry.tlc`, which is the entry point into the Real-Time Workshop target files. For a complete list of available system target files for Real-Time Workshop, see the Real-Time Workshop documentation.

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

Table 4-1: 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 *`model`*`_export.h` 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, *`model`*`.c`, which contains the procedures that implement the model

`srtexport.tlc`
`mrtexport.tlc`
`ertexport.tlc`
`sfcnbody.tlc`
	Static real-time
	Malloc real-time
	Embedded-C
	RTW S-function
		Creates the header file *`model`*`_export.h`, 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 *`model`*`.h`, which defines the data structures used by *`model`*`.c`. The data structures defines include Block Outputs, Parameters, External Inputs and Outputs, and the various work structures. The instances of these structures are declared in *`model`*`.c` (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)

`srtmap.tlc`
`mrtmap.tlc`
`ertmap.tlc`
`sfcnmap.tlc`
	Static real-time
	Malloc real-time
	Embedded-C
	RTW S-function
		Creates the header file *`model`*`.dt`, which contains the mapping information for monitoring block outputs and modifying block parameters

Table 4-1: 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
`sfcnmid.tlc`	RTW S-function	Creates *`model`*`.c`, 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 *`model`*`.prm`, which is included by the *`model`*`.c` file to declare instances of the various data structures defined in *`model`*`.h` (all formats)

`srtreg.tlc`
`mrtreg.tlc`
`ertreg.tlc`
`sfcnreg.tlc`
	Static real-time
	Malloc real-time
	Embedded-C
	RTW S-function
		Creates the source file *`model`*`.h` that is included by the *`model`*`.c` file to satisfy the API (all formats)

`sfcnsid.tlc`	RTW S-function	Creates *`model`*`.c`, which contains data for an RTW S-function.

`srtwide.tlc`
`mrtwide.tlc`
`ertwide.tlc`
`sfcnwide.tlc`
	Static real-time
	Malloc real-time
	Embedded-C
	RTW S-function
		The entry point for code format. This file produces *`model`*`.c`, *`model`*`.h`, and, optionally, *`model`*`.dt`.

Block Target Files

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

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

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

Summary of Target File Usage

In the context of the Real-Time Workshop, there are two types of target files, system target files and block target files:

- System target files

  System target files determine the overall framework of code generation. They determine when blocks get executed, how data gets logged, and so on.
- Block target files

  Block target files determine how each individual block uses its input signals and/or parameters to generate its output or to update its state.

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

- Customize the code generated for a block

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

  Inlining an S-function means writing a target file that tells the Target Language Compiler how to generate code for that S-function block. The Target Language Compiler can automatically generate code for noninlined C MEX S-functions. However, if you inline a C MEX S-function, the compiler can generate more efficient code. Noninlined C MEX S-functions are executed using the S-function Application Program Interface (API) and can be inefficient.

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

- Customize the code generated for all models

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

- Implement support for a new language

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

Refer to Chapter 6, “Directives and Built-in Functions,” for a description of the Target Language and Chapter 3, “A 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** pane of the **Simulation Parameters** dialog box. A close examination of a system target file reveals how code generation occurs. This a listing of the noncomment lines in `grt.tlc`, the target file to generate code for a generic real-time executable:

```
%selectfile NULL_FILE

%assign MatFileLogging = 1
%assign TargetType = "RT"
%assign Language   = "C"

%include "codegenentry.tlc"
```

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

If you want to make changes to modify overall code generation, you must change the system target file. After the initial setup, instead of calling `codegenentry.tlc`, you must call your own TLC files. The code below shows an example system target file called `mygrt.tlc`.

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

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

%include "commonsetup.tlc"

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

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

Generated code is placed in a model or subsystem function. The relevant generated function names and their execution order is detailed in the Real-Time Workshop documentation. During code generation, functions from each of the block target files are executed and the generated code is placed in the appropriate model or subsystem functions.

Block Target Files

Each block has a target file that determines what code should be generated for the block. The code can vary depending on the exact parameters of the block or the types of connections to it (e.g., wide vs. scalar input).

Within each block target file, *block functions* specify the code to be output for the block in the model's or subsystem's `start` function, `output` function, `update` function, and so on.

Block Target File Mapping

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

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

Data Handling with TLC: An Example

Matrix Parameters in Real-Time Workshop

MATLAB, Simulink, and Real-Time Workshop all use column-major ordering for all array storage (1-D, 2-D, ...), so that the “next” element of an array in memory is always accessed by incrementing the first index of the array. For example, all of these element pairs are stored sequentially in memory: `A(i)` and `A(i+1)`, `B(i,j)` and `B(i+1,j)`, `C(i,j,k)` and `C(i+1,j,k)`.For more information on the internal representation of MATLAB data, see “The MATLAB Array” in External Interfaces/API.

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

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

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

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

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

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

```
1   2   3
4   5   6
7   8   9
```

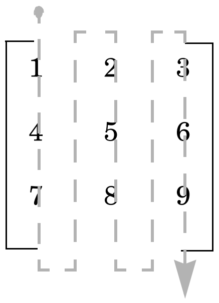
is stored in *`model`*`.rtw` as

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

and results in this definition in *model*.h

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

  [ ... other parameter definitions ... ]

} Parameters;
```

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



```
Parameters 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 accesses matrix parameters via `LibBlockMatrixParameter` and `LibBlockMatrixParameterAddr`, where:

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

`LibBlockMatrixParameterAddr(OutputValues, "", "", 0, "", "", 1)` returns `"&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 `LibBlockParameter` or `LibBlockParameterAddr`.

Contents of model.rtw

The input to the Target Language Compiler is a *model*`.rtw` file, a compilation of *model*`.mdl` that describes blocks, inputs, outputs, parameters, states, storage, and other model components and properties. Here we take a general look at this intermediate file and its principal access methods.

Model.rtw File Overview (p. 5-2)	Identifiers, values, and the structure of records
Using Library Functions to Access model.rtw Contents (p. 5-10)	The safe way to access model records, with one exception

Appendix A, "model.rtw" provides a complete description of the records in *model*`.rtw` files. Please be aware that the structure of this file is very likely to change between releases, which is a very compelling reason to limit your access to *model*`.rtw` to the TLC library functions documented in Chapter 9, "TLC Function Library Reference."

Model.rtw File Overview

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

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

```
ParameterName  value
```

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

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

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

A *record* is specified as

```
RecordName {
      .
      .
}
```

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

```
DataStores {
    NumDataStores    0
}
```

Using Scopes in the model.rtw File

Accessing Values

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

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

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

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

```
CompiledModel.Name
```

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

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

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

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

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

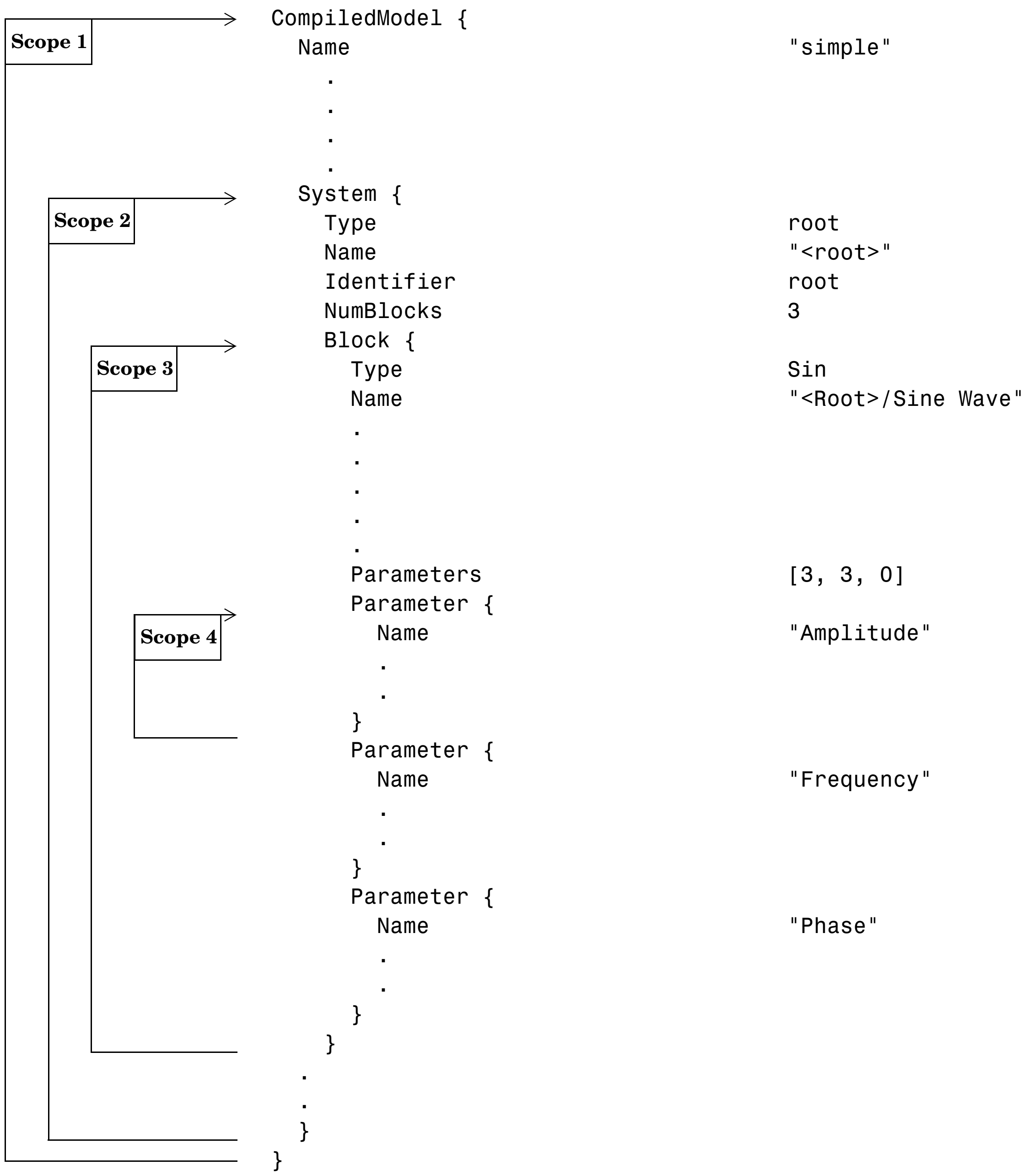
```
%with CompiledModel.System[0].Block[0]
%assign blockName = Name
%endwith
```

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

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

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

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



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

Object Information in the model.rtw File

During code generation, Real-Time Workshop writes information about signal and parameter objects to the *model*.rtw file. An Object record is written for each parameter or signal that meets certain conditions. These conditions are described in “Object Records For Parameters” on page 5-6 and “Object Records For Signals” on page 5-7.

The Object records contain all of the information corresponding to the associated object. To access Object records, you must write Target Language Compiler code (see “Accessing Object Information via TLC” on page 5-8).

Object Records For Parameters

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

1 The parameter resolves to a Simulink.Parameter object (or to a parameter object that comes from a class derived from the Simulink.Parameter class).

2 The parameter’s symbol is preserved in the generated code. The symbol is preserved when:

- **Inline parameters** is on.
- RTWInfo.StorageClass is not set to 'Auto' or 'SimulinkGlobal'.

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

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

Object Records For Signals

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

1 The signal resolves to a `Simulink.Signal` object (or to an object that comes from a class derived from the `Simulink.Signal` class).

2 The signal's symbol is preserved in the generated code. The symbol is preserved if:

- The signal's `RTWInfo.StorageClass` is not set to `'Auto'` or `'SimulinkGlobal'`.
- The signal label is be a valid variable name.
- The signal label is unique throughout the model.

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

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

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

Accessing Object Information via TLC

This section provides sample code to illustrate how to access object information from the *model*.rtw file using TLC code. For more information on TLC and the *model*.rtw file, see Appendix A, "model.rtw".

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

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

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

```
%with CompiledModel.BlockOutputs
  %foreach blockOutputIdx = NumBlockOutputs
    %assign thisBlockOutput = BlockOutput[blockOutputIdx]
    %assign signalName = thisBlockOutput.Identifier
    %if EXISTS("thisBlockOutput.Object.ObjectProperties")
      %with thisBlockOutput.Object.ObjectProperties
        %with RTWInfo.Object.ObjectProperties
          %assign storageClassInObject  = StorageClass
        %endwith \
        %% **********************************\
        %% Access user-defined properties here\
        %% **********************************
        %if EXISTS("MY_PROPERTY_NAME")
          %assign userDefinedPropertyName = MY_PROPERTY_NAME
        %endif
        %% **********************************
      %endwith
    %endif
  %endforeach
%endwith
```

Using Library Functions to Access model.rtw Contents

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

See 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 TLC script ensures that it will be flexible enough to generate code for any instance or configuration of the block, as well as across releases. Exceptions to this do occur, however, as when it is necessary to directly access a field in the block's record. This happens with parameter settings, as discussed in "TLC Code to Access the Parameter Settings" on page 5-11.

Exception to Using the Library Functions

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

mdlRTW Function in C-MEX S-Function Code

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

Resulting Block Record in model.rtw File

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

      ...

      SFcnParamSettings {
        Operator    "AND"
      }
    }
```

TLC Code to Access the Parameter Settings

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

For more details on using parameter settings, see Chapter 8, “Inlining S-Functions.”

Directives and Built-in Functions

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

Compiler Directives (p. 6-2)	The syntax and formats of directives, built-in functions, signal and parameter values, expressions and comments
Command Line Arguments (p. 6-70)	Description of TLC calling arguments, filenames and search paths

Compiler Directives

Syntax

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

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

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

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

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

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

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

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

Table 6-1: Target Language Compiler Directives

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

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description

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

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description

`%if expr`
`%elseif expr`
`%else`
`%endif`
	Conditional inclusion, where the constant-expression *`expr`* must evaluate to an integer. For example, the following code checks whether a parameter, `k`, has the numeric value 0.0 by executing a TLC library function to check for equality.
	
	`%if ISEQUAL(k, 0.0)`
	`  <text and directives to be processed if, k is 0.0>`
	`%endif`
	
	In this and other directives, it is not necessary to expand variables or expressions using the `%<expr>` notation unless *`expr`* appears within a string. For example:
	
	`%if ISEQUAL(idx, “my_idx%<i>”)`, where `idx` and `i` are both strings.
	
	As in other languages, logical evaluations do short circuit (are halted as soon as the result is known).

`%switch expr`
`  %case` *`expr`*
`  %break`
`  %default`
`  %break`
`%endswitch`
	The `switch` directive is very similar to the C language `switch` statement. The expression, `expr`, 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.

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description

`%with`
`%endwith`
	`%with` *`recordName`* is a scoping operator. Use it to bring the named record into the current scope, to remain until the matching `%endwith` is encountered (`%with` directives may be nested as desired).
	
	Note that on the left-hand side of `%assign` statements contained within a `%with` / `%endwith` block, references to fields of records must be fully qualified (see "Assigning Values to Fields of Records" on page 4-16), as in the following example.
	`%with CompiledModel`
	`  %assign oldName = name`
	`  %assign CompiledModel.name = "newname"`
	`%endwith`

`%setcommandswitch string`	Changes the value of a command-line switch as specified by the argument *`string`*. Only the following switches are supported: `v, m, p, O, d, r, I, a` The following example sets the verbosity level to 1. `%setcommandswitch "-v1"` See also "Command Line Arguments" on page 6-70.
`%assert expr`	Tests a value of a Boolean expression. If the expression evaluates to false TLC will issue an error message, a stack trace and exit, and otherwise the execution will be continued as normal. To enable the evaluation of asserts outside the Real-Time Workshop environment, use the command line option "`-da`". When building from within RTW, this flag is not needed and will be ignored, as it is superseded by the **Enable TLC Assertions** check box on the **TLC debugging** section of the **Real-Time Workshop** dialog pane. To control assertion handling from the MATLAB command window, use: `set_param(model, 'TLCAssertion', 'on|off')` to set this flag on or off. Default is Off. `get_param(model, 'TLCAssertion')` to see the current setting.

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description

`%error`
`%warning`
`%trace`
`%exit`
	Flow control directives:
	
	`%error` *`tokens`* — The *`tokens`* are expanded and displayed.
	`%warning` *`tokens`* — The *`tokens`* are expanded and displayed.
	`%trace` *`tokens`* — The *`tokens`* are expanded and displayed only when the "verbose output" command line option `-v` or `-v1` is specified.
	`%exit` *`tokens`* — The *`tokens`* are expanded, displayed, and TLC exits.
	
	Note, when reporting errors, you should use
	
	`%exit Error Message`
	
	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
	
	`%setcommandswitch "-v1" %% force TLC stack trace`
	`%exit Assert message`

`%assign`	Creates identifiers (variables). The general form is
	
	`%assign [::]`*`variable`*` = `*`expression`*
	
	The `::` specifies that the variable being created is a global variable, otherwise, it is a local variable in the current scope (i.e., a local variable in the function).
	
	If you need to format the variable, say, within a string based upon other TLC variables, then you should perform a double evaluation as in
	
	`%assign nameInfo = "The name of this is %<Name>"`
	
	or alternately
	
	`%assign nameInfo = "The name of this is " + Name`
	
	To assign a value to a field of a record you must use a *qualified variable expression*. See "Assigning Values to Fields of Records" on page 4-16.

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description

`%createrecord`	Creates records in memory. This command accepts a list of one or more record specifications (e.g., `{ foo 27 }`). Each record specification contains a list of zero or more name-value pairs (e.g., `foo 27`) that become the members of the record being created. The values themselves can be record specifications, as the following illustrates.
	`%createrecord NEW_RECORD { foo 1 ; SUB_RECORD {foo 2} }`
	`%assign x = NEW_RECORD.foo                    /* x = 1 */`
	`%assign y = NEW_RECORD.SUB_RECORD.foo         /* y = 2 */`
	If more than one record specification follows a given record name, the set of record specifications constitutes an array of records.
	`%createrecord RECORD_ARRAY { foo 1 } ...`
	`                            { foo 2 } ...`
	`                            { bar 3 }`
	`%assign x = RECORD_ARRAY[1].foo               /* x = 2 */`
	`%assign y = RECORD_ARRAY[2].bar               /* y = 3 */`
	Note that arrays of subrecords can be created and indexed by specifying `%createrecord` with identically named subrecords, as follows:
	`%createrecord RECORD_ARRAY { SUB_RECORD { foo 1 } ...`
	`                              SUB_RECORD { foo 2 } ...`
	`                              SUB_RECORD { foo 3 } }`
	`%assign x = RECORD_ARRAY.SUB_RECORD[1].foo  /* x = 2 */`
	`%assign y = RECORD_ARRAY.SUB_RECORD[2].foo  /* y = 3 */`
	If the scope resolution operator (`::`) is the first token after he `%createrecord` token, the record is created in the global scope.

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description

`%addtorecord`	Adds fields to an existing record. The new fields may be name-value pairs or aliases to already existing records.
	
	`%addtorecord OLD_RECORD foo 1`
	
	If the new field being added is a record, then `%addtorecord` will make an alias to that record instead of a deep copy. To make a deep copy, use `%copyrecord`.
	
	`%createrecord NEW_RECORD { foo 1 }`
	`%addtorecord OLD_RECORD NEW_RECORD_ALIAS NEW_RECORD`

`%mergerecord`	Adds (or merges) one or more records into another. The first `record` will contain the results of the merge of the first record plus the contents of 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). `%mergerecord OLD_RECORD NEW_RECORD` If there are duplicate fields in the records being merged the original record's fields will not be overwritten.
`%copyrecord`	Makes a deep copy of an existing record. It creates a new record in a similar fashion to `%createrecord` except the components of the record are deep copied from the existing record. Aliases are replaced by copies. `%copyrecord NEW_RECORD OLD_RECORD`

`%realformat`	Specifies how to format real variables. To format in exponential notation with 16 digits of precision, use
	
	`%realformat "EXPONENTIAL"`
	
	To format without loss of precision and minimal number of characters, use
	
	`%realformat "CONCISE"`
	
	When inlining S-functions, the format is set to `concise`. You can switch to `exponential`, but should switch it back to `concise` when done.

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description
`%language`	This must appear before the first `GENERATE` or `GENERATE_TYPE` function call. This specifies the name of the language as a string, which is being generated as in `%language "C"`. Generally, this is added to your system target file.
`%implements`	Placed within the `.tlc` file for a specific record type, when mapped via `%generatefile`. The syntax is: `%implements "`*`Type`*`" "`*`Language`*`"`. When inlining an S-function in C, this should be the first non-comment line in the file as in `%implements "`*`s_function_name`*`" "C"` The next noncomment lines will be `%function` directives specifying the functionality of the S-function. See the `%language` and `GENERATE` function descriptions for further information.
`%generatefile`	Provides a mapping between a record `Type` and functions contained in a file. Each record can have functions of the same name, but different contents mapped to it (i.e., polymorphism). Generally, this is used to map a `Block` record `Type` to the `.tlc` file that implements the functionality of the block as in `%generatefile "Sin" "sin_wave.tlc"`

Table 6-1: Target Language Compiler Directives (Continued)

Directive	**Description**
`%filescope`	Limits the scope of variables to the file in which they are defined. A `%filescope` directive, anywhere in a file declares that all variables in the file are visible only within that file. Note that this limitation also applies to any files inserted, via the `%include` directive, into the file containing the `%filescope` directive. The `%filescope` directive should not be used within functions or `GENERATE` functions. `%filescope` is useful in conserving memory. Variables whose scope is limited by `%filescope` go out of scope when execution of the file containing them completes. This frees memory allocated to such variables. By contrast, global variables persist in memory throughout execution of the program.

`%include`
`%addincludepath`
	`%include "`*`file`*`.tlc"` — insert specified target file at the current point. Use `%addincludepath "`*`directory`*`"` to add additional paths to be searched. We recommend UNIX-style forward slashes for directory names, as they will work on both UNIX and PC systems. However, if you do use backslashes in PC directory names, be sure to escape them, e.g., `"C:\\mytlc"`. Alternatively, you can express a PC directory name as a literal using the `L` format specifier, as in `L"C:\mytlc"`. All `%include` directives behave as if they were in a global context, such that
	
	`%addincludepath "./sub1"`
	`%addincludepath "./sub2"`
	
	in a `.tlc` file enables either subdirectory to be referenced implicitly:
	
	`%include "file_in_sub1.tlc"`
	`%include "file_in_sub2.tlc"`

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description

`%roll`
`%endroll`
	Multiple inclusion plus intrinsic loop rolling based upon a specified threshold. This directive can be used by most Simulink blocks which have the concept of an overall block width that is usually the width of the signal passing through the block. An example of the `%roll` directive is for a gain operation, `y=u*k`:
	
	`%function Outputs(block, system) Output`
	`  /* %<Type> Block: %<Name> */`
	`  %assign rollVars = ["U", "Y", "P"]`
	`  %roll sigIdx = RollRegions, lcv = RollThreshold, block,...`
	`        "Roller", rollVars`
	`    %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)`
	`    %assign u = LibBlockInputSignal(0, "", lcv, sigIdx)`
	`    %assign k = LibBlockParameter(Gain, "", lcv, sigIdx)`
	`      %<y> = %<u> * %<k>;`
	`  %endroll`
	`%endfunction`
	
	The `%roll` directive is similar to `%foreach`, except it iterates the identifier (`sigIdx` in this example) over roll regions. Roll regions are computed by looking at the input signals and generating regions where the inputs are contiguous. For blocks, the variable `RollRegions` is automatically computed and placed in the `Block` record. An example of a roll regions vector is `[0:19, 20:39]`, where we have two contiguous ranges of signals passing through the block. The first is `0:19` and the second is `20:39`. Each roll region is either placed in a loop body (e.g., the C Language `for` statement), or inlined depending upon whether or not the length of the region is less than the roll threshold.
	
	Each time through the `%roll` loop, `sigIdx` is an integer for the start of the current roll region or an offset relative to the overall block width when the current roll region is less than the roll threshold. The TLC global variable `RollThreshold` is the general model wide value used to decide when to place a given roll region into a loop. When the decision is made to place a given region into a loop, the loop control variable will be a valid identifier (e.g., "`i`"), otherwise it will be `""`.

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description
`%roll` *(continued)*	The `block` parameter is the current block that is being rolled. The `"Roller"` parameter specifies the name for internal `GENERATE_TYPE` calls made by `%roll`. The default `%roll` handler is `"Roller"`, which is responsible for setting up the default block loop rolling structures (e.g., a C `for` loop). The `rollVars` (roll variables) are passed to `"Roller"` functions to create the correct roll structures. The defined loop variables relative to a block are `"U"` All inputs to the block. It assumes you use `LibBlockInputSignal(`*`portIdx`*`, "", lcv, sigIdx)` to access each input, where *`portIdx`* starts at 0 for the first input port. `"U`*`i`*`"` Similar to `"U"`, except only for specific input, *`i`*. `"Y"` All outputs of the block. It assumes you use `LibBlockOutputSignal(`*`portIdx`*`, "", lcv, sigIdx)` to access each output, where *`portIdx`* starts at 0 for the first output port. `"Y`*`i`*`"` Similar to `"Y"`, except only for specific output, *`i`*. `"P"` All parameters of the block. It assumes you use `LibBlockParameter(`*`name`*`, "", lcv, sigIdx)` to access them. `"<param>/`*`name`*`"` Similar to `"P"`, except specific for a specific *`name`*. `RWork` All `RWork` vectors of the block. It assumes you use `LibBlockRWork(`*`name`*`, "", lcv, sigIdx)` to access them. `"<RWork>/`*`name`*`"` Similar to `RWork`, except for a specific *`name`*. `DWork` All `DWork` vectors of the block. It assumes you use `LibBlockDWork(`*`name`*`, "", lcv, sigIdx)` to access them. `"<DWork>/`*`name`*`"` Similar to `DWork`, except for a specific *`name`*. `IWork` All `IWork` vectors of the block. It assumes you use `LibBlockIWork(`*`name`*`, "", lcv, sigIdx)` to access them. `"<IWork>/`*`name`*`"` Similar to `IWork`, except for a specific *`name`*. `PWork` All `PWork` vectors of the block. It assumes you use `LibBlock-PWork(`*`name`*`, "", lcv, sigIdx)` to access them. `"<PWork>/`*`name`*`"` Similar to `PWork`, except for a specific *`name`*. `"Mode"` The mode vector. It assumes you use `LibBlockMode("",lcv,sigIdx)` to access it. `"PZC"` Previous zero crossing state. It assumes you use `LibPrevZCState("",lcv, sigIdx)` to access it.

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description

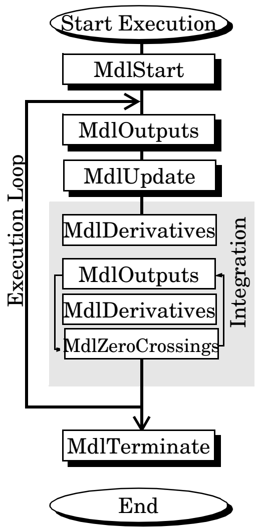
`%roll`
(continued)
	To *roll* your own vector based upon the block's roll regions, you need to walk a pointer to your vector. Assuming your vector is pointed to by the first `PWork`, called *`name`*,
	`datatype *buf = (datatype*)%<LibBlockPWork(name,"","",0)`
	`%roll sigIdx = RollRegions, lcv = RollThreshold, block, ...`
	`      "Roller", rollVars`
	`  *buf++ = whatever;`
	`%endroll`
	Note: In the above example, `sigIdx` and `lcv` are local to the body of the loop.

`%breakpoint`	Sets a breakpoint for the TLC debugger. See "%breakpoint Directive" on page 7-9.

`%function`
`%return`
`%endfunction`
	A function that returns a value is defined as
	`%function name(optional-arguments)`
	`  %return value`
	`%endfunction`
	A void function does not produce any output and is not required to return a value. It is defined as
	`%function name(optional-arguments) void`
	`%endfunction`
	A function that produces outputs to the current stream and is not required to return a value is defined as
	`%function name(optional-arguments) Output`
	`%endfunction`

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description

 Start Execution
MdlStart
MdlOutputs
MdlUpdate
MdlDerivatives
MdlOutputs
MdlDerivatives
MdlZeroCrossings
Integration
Execution Loop
MdlTerminate
End
	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
	
	`%function` **`BlockInstanceSetup`**`(block, system) void`
	Called for each instance of the block within the model.
	`%function` **`BlockTypeSetup`**`(block, system) void`
	Called once for each block type that exists in the model.
	`%function` **`Enable`**`(block, system) Output`
	Use this if the block is placed within an enabled subsystem and has to take specific actions when the subsystem enables. Place within a subsystem enable routine.
	`%function` **`Disable`**`(block, system) Output`
	Use this if the block is placed within a disabled subsystem and has to take specific actions when the subsystem is disabled. Place within a subsystem disable routine.
	`%function` **`Start`**`(block, system) Output`
	Include this function if your block has startup initialization code that needs to be placed within `MdlStart`.

`%function`
`%return`
`%endfunction`
(continued)	

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description

	`%function` **`InitializeConditions`**`(block, system) Output`
	Use this function if your block has state that needs to be initialized at the start of execution and when an enabled subsystem resets states. Place in `MdlStart` and/or subsystem initialization routines.
	`%function` **`Outputs`**`(block, system) Output`
	The primary function of your block. Place in `MdlOutputs`.
	`%function` **`Update`**`(block, system) Output`
	Use this function if your block has actions to be performed once per simulation loop, such as updating discrete states. Place in `MdlUpdate`
	`%function` **`Derivatives`**`(block,system) Output`
	Used this function if your block has derivatives for `MdlDerivatives`.
	`%function` **`ZeroCrossings`**`(block,system) Output`
	Used this function if your block does zero crossing detection and has actions to be performed in `MdlZeroCrossings`.
	`%function` **`Terminate`**`(block, system) Output`
	Use this function if your block has actions that need to be in `MdlTerminate`.

`%foreach`
`%endforeach`
	Multiple inclusion that iterates from `0` to the `upperLimit-1` constant integer expression. Each time through the loop, the loopIdentifier, (e.g., `x`) is assigned the current iteration value.
	
	`%foreach loopIdentifier = upperLimit`
	   `%break    –` *`use this to exit the loop`*
	   `%continue –` *`use this to skip the following code and continue to the next iteration`*
	`%endforeach`
	
	Note: The `upperLimit` expression is cast to a TLC integer value. The `loopIdentifier` is local to the loop body.

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description

`%for`	Multiple inclusion directive with syntax
	`%for ident1 = const-exp1, const-exp2, ident2 = const-exp3`
	`  %body`
	`    %break`
	`    %continue`
	`  %endbody`
	`%endfor`
	
	The first portion of the `%for` directive is identical to the `%foreach` statement. The `%break` and `%continue` directives act the same as they do in the `%foreach` directive. *`const-exp2`* is a Boolean expression that indicates whether the loop should be rolled (see `%roll` above).
	
	If *`const-exp2`* evaluates to `TRUE`, *`ident2`* is assigned the value of *`const-exp3`*. Otherwise, *`ident2`* is assigned an empty string.
	
	Note: `ident1` and `ident2` above are local to the loop body.

`%openfile`
`%selectfile`
`%closefile`
	These are used to manage the files that are created. The syntax is
	`%openfile streamId="filename.ext" mode  {open for writing}`
	`%selectfile streamId                    {select an open file}`
	`%closefile streamId                     {close an open file}`
	
	Note that the "*`filename.ext`*" is optional. If no filename is specified, a variable (string buffer) named *`streamId`* is created containing the output. The mode argument is optional. If specified, it can be `"a"` for appending, `"r"` for reading, or `"w"` for writing.
	
	Note that the special *`streamId`* `NULL_FILE` specifies that no output occur. The special *`streamId`* `STDOUT` specifies output to the terminal.

Table 6-1: Target Language Compiler Directives (Continued)

Directive	Description

	To create a buffer of text, use
	`%openfile buffer`
	`text to be placed in the 'buffer' variable.`
	`%closefile buffer`
	Now buffer contains the expanded text specified between the `%openfile` and `%closefile` directives.

`%generate`	`%generate` *`blk fn`* is equivalent to `GENERATE(`*`blk`*`,`*`fn`*`)`. `%generate` *`blk fn type`* is equivalent to `GENERATE(`*`blk`*`,`*`fn`*`,`*`type`*`)`. See "GENERATE and GENERATE_TYPE Functions" on page 6-35.
`%undef`	`%undef` *`var`* removes the variable *`var`* from scope. If *`var`* is a field in a record, `%undef` removes that field from the record. If *`var`* is a record array, `%undef` removes the first element of the array.

Comments

You can place comments anywhere within a target file. To include comments, use the `/%...%/` or `%%` directives. For example:

```
/%
   Abstract:    Return the field with [width], if field is wide
%/
```

or

```
%endfunction %% Outputs function
```

Use the `/%...%/` construct 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 6-65.

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 6-2: Target Language Values

Value Type String	Example	Description
`"Boolean"`	`1==1`	Result of a comparison or other Boolean operator. The result will be `TLC_TRUE` or `TLC_FALSE`.
`"Complex"`	`3.0+5.0i`	A 64-bit double-precision complex number (`double` on the target machine)
`"Complex32"`	`3.0F+5.0Fi`	A 32-bit single-precision complex number (`float` on the target machine)
`"File"`	`%openfile x`	String buffer opened with `%openfile`
`"File"`	`%openfile x = "out.c"`	File opened with `%openfile`
`"Function"`	`%function foo…`	A user-defined function and `TLC_FALSE` otherwise
`"Gaussian"`	`3+5i`	A 32-bit integer imaginary number (`int` on the target machine)
`"Identifier"`	`abc`	Identifier values can only appear within the `model.rtw` file and cannot appear in expressions (within the context of an expression, identifiers are interpreted as values). To compare against an identifier value, use a string; the identifier will be converted as appropriate to a string.
`"Matrix"`	`Matrix (3,2) [ [ 1, 2]; [3 , 4]; [ 5, 6] ]`	Matrices are simply lists of vectors. The individual elements of the matrix do not need to be the same type, and can be any type except vectors or matrices. The `Matrix (3,2)` text in the example is optional.
`"Number"`	`15`	An integer number (`int` on the target machine)

Table 6-2: Target Language Values (Continued)

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

Target Language Expressions

In any place throughout a target file, you can include an expression of the form `%<expression>`. The Target Language Compiler replaces `%<expression>` with a calculated replacement value based upon the type of the variables within the `%<>` operator. Integer constant expressions are folded and replaced with the resultant value; string constants are concatenated (e.g., two strings in a row `"a" "b"` are replaced with `"ab"`).

```
%<expression>               /* Evaluates the expression.
   * Operators include most standard C
   * operations on scalars. Array indexing
   * is required for certain parameters that
   * are block-scoped within the .rtw file.*/
```

Within the context of an expression, each identifier must evaluate to an identifier or function argument currently in scope. You can use the `%< >` directive on any line to perform textual substitution. To include the > character within a replacement, you must escape it with a "\" character as in

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

The Target Language Expressions table lists the operators that are allowed in expressions. In this table, expressions are listed in order from highest to lowest precedence. The horizontal lines distinguish the order of operations.

As in C expressions, conditional operators are short circuited. If the expression includes a function call with effects, the effects are noticed as if the entire expression was not fully evaluated. For example,

```
%if EXISTS(foo) && foo == 3
```

If the first term of the expression evaluates to a Boolean false (i.e., `foo` does not exist), the second term (`foo == 3`) will not be evaluated.

In the following table, note that "numeric" is one of the following:

- `Boolean`
- `Number`
- `Unsigned`
- `Real`
- `Real32`
- `Complex`
- `Complex32`

- `Gaussian`
- `UnsignedGaussian`

Also, note that "integral" is one of the following:

- `Number`
- `Unsigned`
- `Boolean`

See "TLC Data Promotions" on page 6-26 for information on the promotions that result when the Target Language Compiler operates on mixed types of expressions.

Table 6-3: Target Language Expressions

Expression	Definition
`constant`	Any constant parameter value, including vectors and matrices
`variable-name`	Any valid in-scope variable name, including the local function scope, if any, and the global scope
`::variable-name`	Used within a function to indicate that the function scope is ignored when looking up the variable. This accesses the global scope.
`expr[expr]`	Index into an array parameter. Array indices range from `0` to `N-1`. This syntax is used to index into vectors, matrices, and repeated scope variables.
`expr([expr[,expr]…])`	Function call or macro expansion. The expression outside of the parentheses is the function/macro name; the expressions inside are the arguments to the function or macro. **Note:** Since macros are text-based, they cannot be used within the same expression as other operators.

Table 6-3: Target Language Expressions (Continued)

Expression	Definition
`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.
`expr / expr`	Divide the two expressions; the operands must be numeric.
`expr % expr`	Take the integer modulo of the expressions; the operands must be integral.

Table 6-3: Target Language Expressions (Continued)

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

Table 6-3: Target Language Expressions (Continued)

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

Table 6-3: 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 nonfinite values.

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

TLC Data Promotions

When the Target Language Compiler operates on mixed types of expressions, it promotes the result to the common types indicated in the following table.

This table uses the following abbreviations:

`B`	Boolean
`N`	Number
`U`	Unsigned
`F`	Real32
`D`	Real
`G`	Gaussian
`UG`	UnsignedGaussian
`C32`	Complex32
`C`	Complex

The top row (in bold) and first column (in bold) show the types of expression used in the operation. The intersection of the row and column shows the resulting type of expression.

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

Table 6-4: Datatypes Resulting from Expressions of Mixed Type

	B	**N**	**U**	**F**	**D**	**G**	**UG**	**C32**	**C**
B	B	N	U	F	D	G	UG	C32	C
N	N	N	U	F	D	G	UG	C32	C
U	U	U	U	F	D	UG	UG	C32	C
F	F	F	F	F	D	C32	C32	C32	C
D	D	D	D	D	D	C	C	C	C
G	G	G	UG	C32	C	G	UG	C32	C
UG	UG	UG	UG	C32	C	UG	UG	C32	C
C32	C32	C32	C32	C32	C	C32	C32	C32	C
C	C	C	C	C	C	C	C	C	C

Formatting

By default, the Target Language Compiler outputs all floating-point numbers in exponential notation with 16 digits of precision. To override the default, use the directive

```
%realformat string
```

If *`string`* is `"EXPONENTIAL"`, the standard exponential notation with 16 digits of precision is used. If *`string`* is `"CONCISE"`, the Compiler uses a set of internal heuristics to output the values in a more readable form while maintaining accuracy. The `%realformat` directive sets the default format for `Real` number output to the selected style for the remainder of processing or until it encounters another `%realformat` directive.

Conditional Inclusion

The conditional inclusion directives are

```
%if constant-expression
%else
%elseif constant-expression
%endif
```

and

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

%if

The `constant-expression` must evaluate to an integral expression. It controls the inclusion of all the following lines until it encounters a `%else`, `%elseif`, or `%endif` directive. If the `constant-expression` evaluates to `0`, the lines following the directive are not included. If the `constant-expression` evaluates to any other integral value, the lines following the `%if` directive are included up until the `%endif`, `%elseif`, or `%else` directives.

When the Compiler encounters an `%elseif` directive, and no prior `%if` or `%elseif` directive has evaluated to nonzero, the Compiler evaluates the expression. If the value is 0, the lines following the `%elseif` directive are not included. If the value is nonzero, the lines following the `%elseif` directive are included up until the subsequent `%else`, `%elseif`, or `%endif` directive.

The `%else` directive begins the inclusion of source text if all of the previous `%elseif` statements or the original `%if` statement evaluates to `0`; otherwise, it prevents the inclusion of subsequent lines up to and including the following `%endif`.

The `constant-expression` can contain any expression specified in “Target Language Expressions” on page 6-21.

%switch

The `%switch` statement evaluates the constant expression and compares it to all expressions appearing on `%case` selectors. If a match is found, the body of the `%case` is included; otherwise the `%default` is included.

`%case ... %default` bodies flow together, as in C, and `%break` must be used to exit the switch statement. `%break` will exit the nearest enclosing `%switch`, `%foreach`, or `%for` loop in which it appears. For example,

```
%switch(type)
%case x
   /* Matches variable x. */
   /* Note: Any valid TLC type is allowed. */
%case "Sin"
   /* Matches Sin or falls through from case x. */
   %break
   /* Exits the switch. */
%case "gain"
   /* Matches gain. */
   %break
%default
   /* Does not match x, "Sin," or "gain." */
%endswitch
```

In general, this is a more readable form for the `%if/%elseif/%else` construction.

Multiple Inclusion

%foreach

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

```
%foreach identifier = constant-expression
   %break
   %continue
%endforeach
```

The `constant-expression` must evaluate to an integral expression, which then determines the number of times to execute the `foreach` loop. The `identifier` increments from `0` to one less than the specified number. Within the `foreach` loop, you can use `x`, where `x` is the identifier, to access the identifier variable. `%break` and `%continue` are optional directives that you can include in the `%foreach` directive:

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

%for

Note The `%for` directive is functional, but it is not recommended. Rather, use `%roll`, which provides the same capability in a more open way. The Real-Time Workshop does not make use of the `%for` construct.

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

```
%for ident1 = const-exp1, const-exp2, ident2 = const-exp3
  %body
    %break
    %continue
  %endbody
%endfor
```

The first portion of the `%for` directive is identical to the `%foreach` statement in that it causes a loop to execute from `0` to `N-1` times over the body of the loop. In the normal case, it includes only the lines between `%body` and `%endbody`, and the lines between the `%for` and `%body`, and ignores the lines between the `%endbody` and `%endfor`.

The `%break` and `%continue` directives act the same as they do in the `%foreach` directive.

`const-exp2` is a Boolean expression that indicates whether the loop should be rolled. If `const-exp2` is true, *`ident2`* receives the value of `const-exp3`; otherwise it receives the null string. When the loop is rolled, all of the lines between the `%for` and the `%endfor` are included in the output exactly one time. *`ident2`* specifies the identifier to be used for testing whether the loop was rolled within the body. For example:

```
%for Index = <NumNonVirtualSubsystems>3, rollvar="i"
```

```
        {
            int i;

            for (i=0; i< %<NumNonVirtualSubsystems>; i++)
            {
              %body
      x[%<rollvar>] = system_name[%<rollvar>];
              %endbody

            }
        }
  %endfor
```

If the number of nonvirtual subsystems (`NumNonVirtualSubsystems`) is greater than or equal to 3, the loop is rolled, causing all of the code within the loop to be generated exactly once. In this case, `Index = 0`.

If the loop is not rolled, the text before and after the body of the loop is ignored and the body is generated `NumNonVirtualSubsystems` times.

This mechanism gives each individual loop control over whether or not it should be rolled.

%roll

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

```
  %roll ident1 = roll-vector-exp, ident2 = threshold-exp, ...
                 block-exp [, type-string [,exp-list] ]
    %break
    %continue
  %endroll
```

This statement uses the `roll-vector-exp` to expand the body of the `%roll` statement multiple times as in the `%foreach` statement. If a range is provided in the `roll-vector-exp` and that range is larger than the `threshold-exp` expression, the loop will roll. When a loop rolls, the body of the loop is expanded once and the identifier (`ident2`) provided for the threshold expression is set to the name of the loop control variable. If no range is larger than the specified rolling threshold, this statement is identical in all respects to the `%foreach` statement.

For example,

```
%roll Idx = [ 1 2 3:5, 6, 7:10 ], lcv = 10, ablock
%endroll
```

In this case, the body of the `%roll` statement expands 10 times as in the `%foreach` statement since there are no regions greater than or equal to 10. `Idx` counts from 1 to 10, and `lcv` is set to the null string, `""`.

When the Target Language Compiler determines that a given block will roll, it performs a `GENERATE_TYPE` function call to output the various pieces of the loop (other than the body). The default type used is `Roller`; you can override this type with a string that you specify. Any extra arguments passed on the `%roll` statement are provided as arguments to these special-purpose functions. The called function is one of these four functions.

RollHeader(block, ...). This function is called once on the first section of this roll vector that will actually roll. It should return a string that is assigned to the `lcv` within the body of the `%roll` statement.

LoopHeader(block, StartIdx, Niterations, Nrolled, ...). This function is called once for each section that will roll prior to the body of the `%roll` statement.

LoopTrailer(block, Startidx, Niterations, Nrolled, ...). This function is called once for each section that will roll after the body of the `%roll` statement.

RollTrailer(block, ...). This function is called once at the end of the `%roll` statement if any of the ranges caused loop rolling.

These functions should output any language-specific declarations, loop code, and so on as required to generate correct code for the loop. An example of a `Roller.tlc` file is

```
%implements Roller "C"
%function RollHeader(block) Output
   {
       int i;
   %return ("i")
%endfunction

%function LoopHeader(block,StartIdx,Niterations,Nrolled) Output
   for (i = %<StartIdx>; i < %<Niterations+StartIdx>; i++)
   {
%endfunction
%function LoopTrailer(block,StartIdx,Niterations,Nrolled) Output
   }
%endfunction

%function RollTrailer(block) Output
   }
%endfunction
```

Note The Target Language Compiler function library provided with Real-Time Workshop has the capability to extract references to the block I/O and other Real-Time Workshop vectors that vastly simplify the body of the `%roll` statement. These functions include `LibBlockInputSignal`, `LibBlockOutputSignal`, `LibBlockParameter`, `LibBlockRWork`, `LibBlockIWork`, `LibBlockPWork`, and `LibDeclareRollVars`, `LibBlockMatrixParameter`, `LibBlockParameterAddr`, `LibBlockContinuousState`, and `LibBlockDiscreteState` function reference pages in Chapter 9, "TLC Function Library Reference." This library also includes a default implementation of `Roller.tlc` an a "flat" roller.

Extending the former example to a loop that rolls

```
%language "C"
%assign ablock = BLOCK { Name "Hi" }
%roll Idx = [ 1:20, 21, 22, 23:25, 26:46], lcv = 10, ablock
   Block[%< lcv == "" ? Idx : lcv>] *= 3.0;
%endroll
```

This Target Language Compiler code produces this output:

```
{
   int            i;
   for (i = 1; i < 21; i++)
   {
      Block[i] *= 3.0;
   }
   Block[21] *= 3.0;
   Block[22] *= 3.0;
   Block[23] *= 3.0;
   Block[24] *= 3.0;
```

```
        Block[25] *= 3.0;
        for (i = 26; i < 47; i++)
        {
            Block[i] *= 3.0;
        }
    }
```

Object-Oriented Facility for Generating Target Code

The Target Language Compiler provides a simple object-oriented facility. The language directives are

```
%language string
%generatefile
%implements
```

This facility was designed specifically for customizing the code for Simulink blocks, but can be used for other purposes as well.

%language

The `%language` directive specifies the target language being generated. It is required as a consistency check to ensure that the correct implementation files are found for the language being generated. The `%language` directive must appear prior to the first `GENERATE` or `GENERATE_TYPE` built-in function call. `%language` specifies the language as a string. For example,

```
%language "C"
```

All blocks in Simulink have a `Type` parameter. This parameter is a string that specifies the type of the block, e.g., `"Sin"` or `"Gain"`. The object-oriented facility uses this type to search the path for a file that implements the correct block. By default the name of the file is the `Type` of the block with `.tlc` appended, so for example, if the `Type` is `"Sin"` the Compiler would search for `"Sin.tlc"` along the path. You can override this default filename using the `%generatefile` directive to specify the filename that you want to use to replace the default filename. For example,

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

The files that implement the block-specific code must contain a `%implements` directive indicating both the type and the language being implemented. The

Target Language Compiler will produce an error if the `%implements` directive does not match as expected. For example,

```
%implements "Sin" "Pascal"
```

causes an error if the initial language choice was C.

You can use a single file to implement more than one target language by specifying the desired languages in a vector. For example,

```
%implements "Sin" "C"
```

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

```
%implements * "C"
```

Note The use of the wildcard (`*`) is not recommended because it relaxes error checking for the `%implements` directive.

GENERATE and GENERATE_TYPE Functions

The Target Language Compiler has two built-in functions that dispatch object-oriented calls, `GENERATE` and `GENERATE_TYPE`. You can call any function appearing in an implementation file (from outside the specified file) only by using the `GENERATE` and `GENERATE_TYPE` special functions.

GENERATE. The `GENERATE` function takes two or more input arguments. The first argument must be a valid scope and the second a string containing the name of the function to call. The `GENERATE` function passes the first block argument and any additional arguments specified to the function being called. The return argument is the value (if any) returned from the function being called. Note that the Compiler automatically "scopes" or adds the first argument to the list of scopes searched as if it appears on a `%with` directive line. (See "Variable Scoping" on page 6-55.) This scope is removed when the function returns.

GENERATE_TYPE. The `GENERATE_TYPE` function takes three or more input arguments. It handles the first two arguments identically to the `GENERATE` function call. The third argument is the type; the type specified in the Simulink block is ignored. This facility is used to handle S-function code generation by the Real-Time Workshop. That is, the block type is `S-function`, but the Target

Language Compiler generates it as the specific S-function specified by GENERATE_TYPE. For example,

```
GENERATE_TYPE(block, "Output", "dp_read")
```

specifies that S-function block is of type dp_read.

The block argument and any additional arguments are passed to the function being called. Similar to the GENERATE built-in function, the Compiler automatically scopes the first argument before the GENERATE_TYPE function is entered and then removes the scope on return.

Within the file containing %implements, function calls are looked up first within the file and then in the global scope. This makes it possible to have hidden helper functions used exclusively by the current object.

Note It is not an error for the GENERATE and GENERATE_TYPE directives to find no matching functions. This is to prevent requiring empty specifications for all aspects of block code generation. Use the GENERATE_FUNCTION_EXISTS or GENERATE_TYPE_FUNCTION_EXISTS directives to determine if the specified function actually exists.

Output File Control

The structure of the output file control construct is

```
%openfile string optional-equal-string optional-mode
%closefile id
%selectfile id
```

%openfile

The %openfile directive opens a file or buffer for writing; the required string variable becomes a variable of type file. For example,

```
%openfile x                /% Opens and selects x for writing. %/
%openfile out = "out.h"    /% Opens "out.h" for writing. %/
```

%selectfile

The `%selectfile` directive selects the file specified by the variable as the current output stream. All output goes to that file until another file is selected using `%selectfile`. For example,

```
%selectfile x               /% Select file x for output. %/
```

%closefile

The `%closefile` directive closes the specified file or buffer, and if this file is the currently selected stream, `%closefile` invokes `%selectfile` to reselect the last previously selected output stream.

There are two possible cases that `%closefile` must handle:

- If the stream is a file, the associated variable is removed as if by `%undef`.
- If the stream is a buffer, the associated variable receives all the text that has been output to the stream. For example,

```
%assign x = ""                    /% Creates an empty string. %/
%openfile x
"hello, world"
%closefile x /% x = "hello, world\n"%/
```

If desired, you can append to an output file or string by using the optional mode, a, as in

```
%openfile "foo.c", "a"            %% Opens foo.c for appending.
```

Input File Control

The input file control directives are

```
%include string
%addincludepath string
```

%include

The `%include` directive searches the path for the target file specified by `string` and includes the contents of the file inline at the point where the `%include` statement appears.

%addincludepath

The `%addincludepath` directive adds an additional include path to be searched when the Target Language Compiler references `%include` or block target files. The syntax is

```
%addincludepath string
```

The `string` can be an absolute path or an explicit relative path. For example, to specify an absolute path, use

```
%addincludepath "C:\\directory1\\directory2"      (PC)
%addincludepath "/directory1/directory2"          (UNIX)
```

To specify a relative path, the path must explicitly start with "`.`". For example,

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

When an explicit relative path is specified, the directory that is added to the Target Language Compiler search path is created by concatenating the location of the target file that contains the `%addincludepath` directive and the explicit relative path.

The Target Language Compiler searches the directories in the following order for target or include files:

1 The current directory

2 Any `%addincludepath` directives

3 Any include paths specified at the command line via `-I`

Typically, `%addincludepath` directives should be specified in your system target file. Multiple `%addincludepath` directives will add multiple paths to the Target Language Compiler search path.

Asserts, Errors, Warnings, and Debug Messages

The related assert, error, warning, and debug message directives are

```
%assert expression
%error tokens
%warning tokens
```

```
%trace tokens
%exit tokens
```

These directives produce error, warning, or trace messages whenever a target file detects an error condition, or tracing is desired. All of the tokens following the directive on a line become part of the generated error or warning message.

The Target Language Compiler places messages generated by `%trace` onto `stderr` if and only if you specify the verbose mode switch (`-v`) to the Target Language Compiler. See "Command Line Arguments" on page 6-70 for additional information about switches.

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

Note In order for `%assert` directives to be evaluated, **Enable TLC Assertions** must be selected on the **TLC debugging** section of the **Real-Time Workshop dialog**. The default action is for asserts not to be evaluated.

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

Built-In Functions and Values

Table 6-5, 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 6-5: TLC Built-in Functions and Values

Built-In Function Name	**Expansion**
`CAST(expr, expr)`	The first expression must be a string that corresponds to one of the type names in the 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 `CAST("Real", *variable-name*)` 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 `%realformat`).
`EXISTS(var)`	If the `var` identifier is not currently in scope, the result is `TLC_FALSE.` If the identifier is in scope, the result is `TLC_TRUE`. `var` can be a single identifier or an expression involving the `.` and `[]` operators. **Note:** prior to TLC release 4, the semantics of `EXISTS` differed from the above. See "Compatibility Issues" on page 1-19.
`FEVAL(matlab-command, TLC-expressions)`	Performs an evaluation in MATLAB. For example `%assign result = FEVAL("sin",3.14159)` The `%matlab` directive can be used to call a MATLAB function that does not return a result. For example, `%matlab disp(2.718)` **Note:** if the MATLAB function returns more than one value, TLC only receives the first value.
`FILE_EXISTS(expr)`	`expr` must be a string. If a file by the name `expr` does not exist on the path, the result is `TLC_FALSE`. If a file by that name exists on the path, the result is `TLC_TRUE`.

Table 6-5: TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
`FORMAT(`*`realvalue`*`, `*`format`*`)`	The first expression is a `Real` value to format. The second expression is either "`EXPONENTIAL`" or "`CONCISE`". Outputs the `Real` value in the designated format where `EXPONENTIAL` uses exponential notation with 16 digits of precision, and `CONCISE` outputs the number in a more readable format while maintaining numerical accuracy.
`FIELDNAMES(`*`record`*`)`	Returns a array of strings containing the record field names associated with the record. As it returns a sorted list of strings, function is O(n*log(n)).
`GETFIELD(`*`record`*`, `*`“fieldname”`*`)`	Returns the contents of the specified field name, if the field name is associated with the record. By virtue of hash lookup, executes in constant time
`GENERATE(`*`record`*`, `*`function-name`*`, ...)`	This is used to execute function calls mapped to a specific record type (i.e., Block record functions). For example, use this to execute the functions in the `.tlc` files for built-in blocks. Note, TLC automatically “scopes” or adds the first argument to the list of scopes searched as if it appears on a `%with` directive line.
`GENERATE_FILENAME(`*`type`*`)`	For the specified record type, does a `.tlc` file exist? Use this to see if the `GENERATE_TYPE` call will succeed.
`GENERATE_FORMATTED_VALUE (expr, string)`	Returns a potentially multiline string that can be used to declare the value(s) of `expr` in the current target language. The second argument is a string that is used as the variable name in a descriptive comment on the first line of the return string. If the second argument is the empty string, `""`, then no descriptive comment is put into the output string.

Table 6-5: TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
`GENERATE_FUNCTION_EXISTS (`*`record`*`, `*`function-name`*`)`	Determines if a given block function exists. The first expression is the same as the first argument to `GENERATE`, namely a block scoped variable containing a `Type`. The second expression is a string that should match the function name.
`GENERATE_TYPE (`*`record`*`, `*`function-name`*`, `*`type`*`, ...)`	Similar to `GENERATE`, except *`type`* overrides the `Type` field of the record. Use this when executing functions mapped to specific S-function blocks records based upon the S-function name (i.e., name becomes type).
`GENERATE_TYPE_FUNCTION_EXISTS (`*`record`*`, `*`function-name`*`, `*`type`*`)`	Same as `GENERATE_FUNCTION_EXISTS` except it overrides the `Type` built into the record.
`GET_COMMAND_SWITCH`	Returns the value of command-line switches. Only the following switches are supported: `v, m, p, O, d, r, I, a` See also “Command Line Arguments” on page 6-70
`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 6-5: 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(`*`expr1`*`,` *`expr2`*`)`	Where the datatypes of both expressions are numeric: returns `TLC_TRUE` if the first expression contains the same value as the second expression; returns `TLC_FALSE` otherwise. Where the datatype of either expression is non- numeric (e.g. string or record): returns `TLC_TRUE` if and only if both expressions have the same datatype and contain the same value; returns `TLC_FALSE` otherwise.
`ISEMPTY(expr)`	Returns `TLC_TRUE` if the expression contains an empty string, vector, or record, and `TLC_FALSE` otherwise.
`ISFIELD(record, “fieldname”)`	Returns `TLC_TRUE` if the field name is associated to the record, and `TLC_FALSE` otherwise.
`ISINF(expr)`	Returns `TLC_TRUE` if the value of the expression is `inf` and `TLC_FALSE` otherwise.
`ISNAN(expr)`	Returns `TLC_TRUE` if the value of the expression is `NAN`, and `TLC_FALSE` otherwise.
`ISFINITE(expr)`	Returns `TLC_TRUE` if the value of the expression is not +/- `inf` or `NAN,` and `TLC_FALSE` otherwise.
`NULL_FILE`	A predefined file for no output that you can use as an argument to `%selectfile` to prevent output.
`NUMTLCFILES`	The number of target files used thus far in expansion.

Table 6-5: TLC Built-in Functions and Values (Continued)

Built-In Function Name	Expansion
`OUTPUT_LINES`	Returns the number of lines that have been written to the currently selected file or buffer. Does not work for `STDOUT` or `NULL_FILE`
`REAL(expr)`	Returns the real part of a complex number.
`REMOVEFIELD(record, “fieldname”)`	Removes the specified field from the contents of the record. Returns `TLC_TRUE` if the field was removed; otherwise returns `TLC_FALSE`.
`ROLL_ITERATIONS()`	Returns the number of times the current roll regions are looping or `NULL` if not inside a `%roll` construct.
`SETFIELD(record, “fieldname”, value)`	Sets the contents of the field name associated to the record. Returns `TLC_TRUE` if the field was added; otherwise returns `TLC_FALSE`.
`SIZE(expr[,expr])`	Calculates the size of the first expression and generates a two-element, row vector. If the second operand is specified, it is used as an integral index into this row vector; otherwise the entire row vector is returned. `SIZE(x)` applied to any scalar returns `[1 1]`. `SIZE(x)` applied to any scope returns the number of repeated entries of that scope type (e.g., `SIZE(Block)` returns `[1,<number of blocks>]`.
`SPRINTF(format,var,...)`	Formats the data in variable `var` (and in any additional variable arguments) under control of the specified format string, and returns a string variable containing the values. Operates like C library `sprintf()`, except that output is the return value rather than contained in an argument to `sprintf`.
`STDOUT`	A predefined file for `stdout` output. You can use this as an argument to `%selectfile` to force output to `stdout`.

Table 6-5: TLC Built-in Functions and Values (Continued)

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

Table 6-5: TLC Built-in Functions and Values (Continued)

Built-In Function Name	**Expansion**
`TLC_TIME`	The date and time of compilation.
`TLC_VERSION`	The version and date of the Target Language Compiler.
`TYPE(expr)`	Evaluates `expr` and determines the result type. The result of this function is a string that corresponds to the type of the given expression. See value type string in Table 6-2, Target Language Values for possible values.
`UINT8MAX`	`255U`
`UINT16MAX`	`65535U`
`UINT32MAX`	`4294967295U`
`UINTMAX`	Maximum unsigned integer value on target machine.
`WHITE_SPACE(expr)`	Accepts a string and returns 1 if the string contains only whitespace characters (, `\t`, `\n`, `\r`); returns 0 otherwise.
`WILL_ROLL(expr1, expr2)`	The first expression is a roll vector and the second expression is a roll threshold. This function returns `true` if the vector contains a range that will roll.

FEVAL Function

The `FEVAL` built-in function calls MATLAB M-file functions and MEX-functions. The structure is

```
%assign result = FEVAL( matlab-function-name, rhs1, rhs2, ...
   rhs3, ... );
```

Note Only a single left-side argument is allowed when calling MATLAB.

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

Table 6-6:

TLC Type	MATLAB Type
`"Boolean"`	`Boolean` (scalar or Matrix)
`"Number"`	`Double` (scalar or Matrix)
`"Real"`	`Double` (scalar or Matrix)
`"Real32"`	`Double` (scalar or Matrix)
`"Unsigned"`	`Double` (scalar or Matrix)
`"String"`	`String`
`"Vector"`	If the vector is homogeneous, it will convert to a MATLAB vector of the appropriate value. If the vector is heterogeneous, it converts to a MATLAB cell array.
`"Gaussian"`	`Complex` (scalar or Matrix)
`"UnsignedGaussian"`	`Complex` (scalar or Matrix)
`"Complex"`	`Complex` (scalar or Matrix)
`"Complex32"`	`Complex` (scalar or Matrix)
`"Identifier"`	`String`
`"Subsystem"`	`String`
`"Range"`	expanded vector of `Doubles`
`"Idrange"`	expanded vector of `Doubles`
`"Matrix"`	If the matrix is homogeneous, it will convert to a MATLAB matrix of the appropriate value. If the matrix is heterogeneous, it converts to a MATLAB cell array. (Cell arrays can be nested.)
`"Scope" or "Record"`	`Structure with elements`

Table 6-6:

TLC Type	MATLAB Type
`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, nor is conversion or downcast of 64-bit integer values,

Table 6-7:

MATLAB Type	TLC Type
`String`	`String`
`Vector of Strings`	`Vector of Strings`
`Boolean (scalar or Matrix)`	`Boolean` (scalar or Matrix)
`INT8,INT16,INT32 (scalar or Matrix)`	`Number` (scalar or Matrix)
`INT64`	Not supported
`UINT64`	Not supported
`Complex INT8,INT16,INT32 (scalar or Matrix)`	Gaussian (scalar or Matrix)
`UINT8,UINT16,UINT32 (scalar or Matrix)`	`Unsigned` (scalar or Matrix)
`Complex UINT8,UINT16,UINT32 (scalar or Matrix)`	`UnsignedGaussian` (scalar or Matrix)
`Single precision`	`Real32` (scalar or Matrix)
`Complex single precision`	`Complex32` (scalar or Matrix)

Table 6-7:

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

Other value types are not currently supported.

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

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

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

Constant Form	TLC Type
`1.0`	`"Real"`
`1.0[F/f]`	`"Real32"`
`1`	`"Number"`
`1[U|u]`	`"Unsigned"`
1.0i	`"Complex"`
`1[Ui|ui]`	`"UnsignedGaussian"`

Constant Form	TLC Type
`1i`	`"Gaussian"`
`1.0[Fi|fi]`	`"Complex32"`

Note The suffix controls the Target Language Compiler type obtained from the constant.

This table shows Target Language Compiler constants and their equivalent MATLAB values.

TLC Constant(s)	Equivalent MATLAB Value
`rtInf`, `Inf`, `inf`	`+inf`
`rtMinusInf`	`-inf`
`rtNan`, `NaN`, `nan`	`nan`
`rtInfi`, `Infi`, `infi`	`inf*i`
`rtMinusInfi`	`-inf*i`
`rtNaNi`, `NaNi`, `nani`	`nan*i`

TLC Reserved Constants

For double precision values, the following are defined for infinite and not-a-number IEEE values

`rtInf`, `inf`, `rtMinusInf`, `-inf`, `rtNaN`, `nan`

and their corresponding version when complex

`rtInfi`, `infi`, `rtMinusInfi`, `-infi`, `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 side can be a qualified reference to a variable using the `.` and `[]` operators, or it can be a single element of a vector or matrix. In the case of the matrix, only the single element is changed by the assignment.

The `%assign` directive inserts new identifiers into the local function scope (if any), file function scope (if any), generate file scope (if any), or into the global scope. Identifiers introduced into the function scope are not available within functions being called, and are removed upon return from the function. Identifiers inserted into the global scope are persistent. Existing identifiers can be changed by completely respecifying them. The constant expressions can include any legal identifiers from the `.rtw` files. You can use `%undef` to delete identifiers in the same way that you use it to remove macros.

Within the scope of a function, variable assignments always create new local variables unless you use the `::` scope resolution operator. For example, given a local variable `foo` and a global variable `foo`

```
%function …
…
%assign foo = 3
…
%endfunction
```

In this example, the assignment always creates a variable `foo` local to the function that will disappear when the function exits. Note that `foo` is created even if a global `foo` already exists.

In order to create or change values in the global scope, you must use the scope resolution operator (`::`) to disambiguate, as in

```
%function …
%assign foo = 3
%assign ::foo = foo
…
%endfunction
```

The scope resolution operator (`::`) forces the compiler to assign to the global `foo`, or to change its existing value to 3.

Note It is an error to change a value from the Real-Time Workshop file without qualifying it with the scope. This example does not generate an error:

```
%assign CompiledModel.name = "newname"  %% No error
```

This example generates an error:

```
%with CompiledModel
  %assign name = "newname"              %% Error
%endwith
```

Creating Records

Use the `%createrecord` directive to build new records in the current scope. For example, if you want to create a new record called `Rec` that contains two items (e.g., `Name "Name"` and `Type "t"`), use

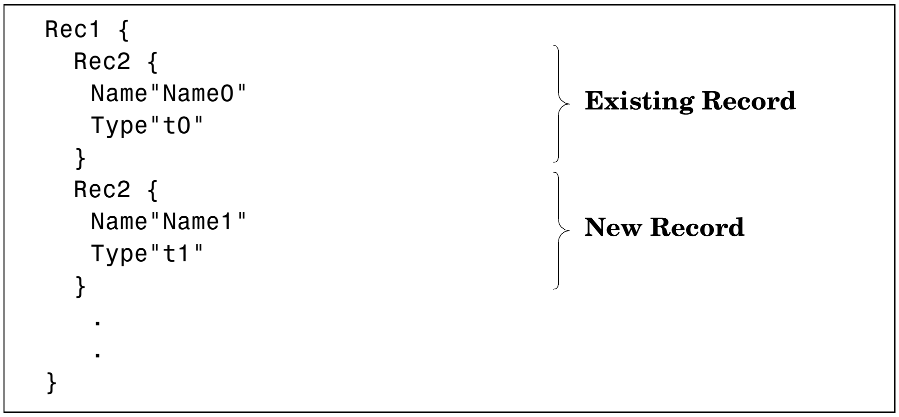
```
%createrecord Rec {  Name "Name"; Type "t" }
```

Adding Records

Use the `%addtorecord` directive to add new records to existing records. For example, if you have a record called `Rec1` that contains a record called `Rec2`, and you want to add an additional `Rec2` to it, use

```
%addtorecord Rec1 Rec2 {  Name "Name1"; Type "t1" }
```

This figure shows the result of adding the record to the existing one.



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

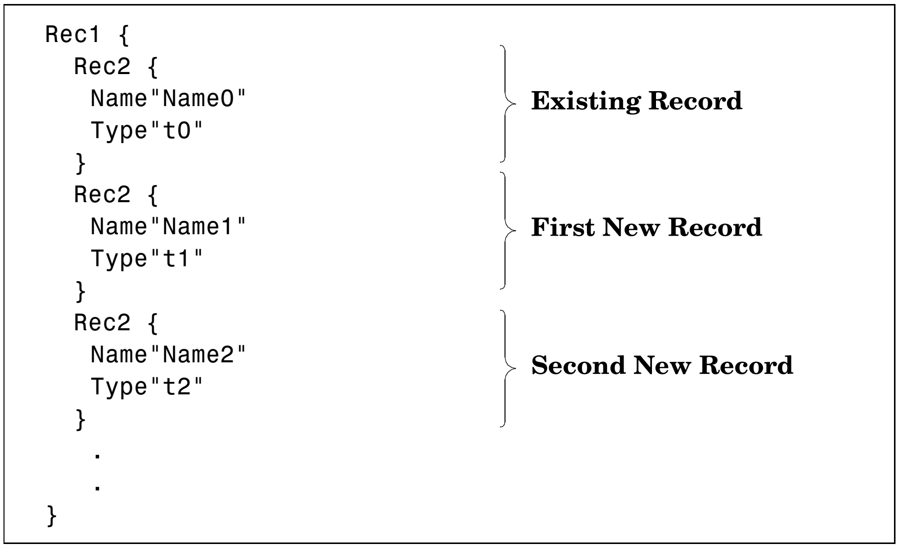
If you want to access the new record, you can use

```
%assign myname = Rec1.Rec2[1].Name
```

In this same example, if you want to add two records to the existing record, use

```
%addtorecord Rec1 Rec2 {  Name "Name1"; Type "t1" }
%addtorecord Rec1 Rec2 {  Name "Name2"; Type "t2" }
```

This produces



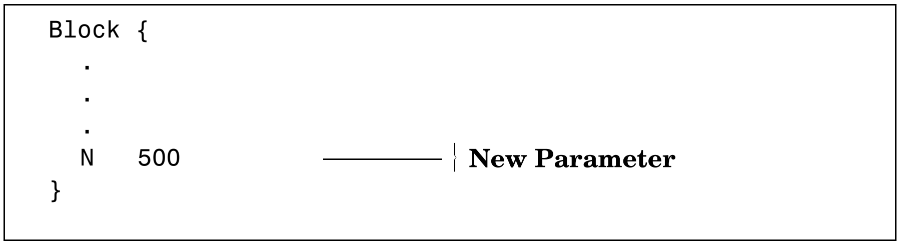
```
Rec1 {
  Rec2 {
   Name"Name0"
   Type"t0"
  }
  Rec2 {
   Name"Name1"
   Type"t1"
  }
  Rec2 {
   Name"Name2"
   Type"t2"
  }
   .
   .
}
```

Adding Parameters to an Existing Record

You can use the `%assign` directive to add a new parameter to an existing record. For example,

```
%addtorecord Block[Idx] N 500 /% Adds N with value 500 to Block %/
%assign myn = Block[Idx].N    /% Gets the value 500 %/
```

adds a new parameter, `N`, at the end of an existing block with the name and current value of an existing variable as shown in this figure. It returns the block value.



```
Block {
  .
  .
  .
  N   500
}
```

Variable Scoping

This section discusses how the Target Language Compiler resolves references to variables (including records).

Scope, in this document, has two related meanings. First, scope is an attribute of a variable that defines its visibility and persistence. For example, a variable defined within the body of a function is visible only within that function, and it persists only as long as that function is executing. Such a variable has *function (or local) scope*. Each TLC variable has one (and only one) of the scopes described in "Scopes" below.

The term scope also refers to a collection, or *pool*, of variables that have the same scope. At any point in the execution of a TLC program, several scopes may exist. For example, during execution of a function, a function scope (the pool of variables local to the function) exists. In all cases, a global scope (the pool of global variables) would also exist.

To resolve variable references, TLC maintains a search list of current scopes and searches them in a well-defined sequence. The search sequence is described in "How TLC Resolves Variable References" on page 6-60.

Dynamic scoping refers to the process by which TLC creates and deallocates variables and the scopes in which they exist. For example, variables in a function scope exist only while the defining function executes.

Scopes

The following sections describe the possible scopes that a TLC variable can have.

Global Scope. By default, TLC variables have global scope. Global variables are visible to, and can be referenced by, code anywhere in a TLC program. Global variables persist throughout the execution of the TLC program. Global variables are said to belong to the *global pool*.

Note in particular that the `CompiledModel` record of the `model.rtw` file has global scope. Therefore, you can access this structure from any of your TLC functions or files.

You can use the scope resolution operator (`::`) to explicitly reference or create global variables from within a function. See "The Scope Resolution Operator" on page 6-60 for examples.

Note that you can use the `%undef` directive to free up memory used by global variables.

File Scope. Variables with file scope are visible only within the file in which they are created. To limit the scope of variables in this way, use the `%filescope` directive anywhere in the defining file.

In the following code fragment, the variables `fs1` and `fs2` have file scope. Note that the `%filescope` directive does not have to be positioned before the statements that create the variables:

```
%assign fs1 = 1
%filescope
%assign fs2 = 3
```

Variables whose scope is limited by `%filescope` go out of scope when execution of the file containing them completes. This lets you free up memory allocated to such variables.

Function (Local) Scope. Variables defined within the body of a function have function scope. That is, they are visible within and local to the defining function. For example, in the following code fragment, the variable `localv` is local to the function `foo`. The variable `x` is global:

```
%assign x = 3

%function foo(arg)
   %assign localv = 1
   %return x + localv
%endfunction
```

A local variable can have the same name as a global variable. To refer, within a function, to identically-named local and global variables, you must use the scope resolution operator (`::`) to disambiguate the variable references. See "The Scope Resolution Operator" on page 6-60 for examples.

Note Functions themselves (as opposed to the variables defined within functions) have global scope. There is one exception: functions defined in generate scope are local to that scope. See "Generate Scope" on page 6-57.

%with Scope. The `%with` directive adds a new scope, referred to as a `%with` *scope*, to the current list of scopes. This directive makes it easier to refer to block-scoped variables.

The structure of the `%with` directive is

```
%with expression
%endwith
```

For example, the directive

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

adds the `CompiledModel.System[sysidx]` scope to the search list. This scope is searched before anything else. You can then refer to the system name simply by

```
Name
```

instead of

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

Generate Scope. *Generate* scope is a special scope used by certain built-in functions that are designed to support code generation. These functions dispatch function calls that are mapped to a specific record type. This capability supports a type of polymorphism in which different record types are associated with functions (analogous to methods) of the same name. Typically, this feature is used to map `Block` records to functions that implement the functionality of different block types.

Functions that employ generate scope include `GENERATE`, `GENERATE_TYPE`, `GENERATE_FUNCTION_EXISTS`, and `GENERATE_TYPE_FUNCTION_EXISTS` (See "GENERATE and GENERATE_TYPE Functions" on page 6-35). This section will discuss generate scope using the `GENERATE` built-in function as an example.

The syntax of the `GENERATE function is`

```
GENERATE(blk,fn)
```

The first argument (*`blk`*) to `GENERATE` is a valid record name. The second argument (*`fn`*) is the name of a function to be dispatched. When a function is dispatched through a `GENERATE` call, TLC automatically adds *`blk`* to the list of scopes that is searched when resolving variable references. Thus the record (*blk*) is visible to the dispatched function, as if there were an implicit `%with <blk>... %endwith` directive in the dispatched function.

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

Three TLC files, demonstrating the use of generate scope, are listed below. The file `polymorph.tlc` creates two records representing two hypothetical block types, `MyBlock` and `YourBlock`. Each record type has an associated function named `aFunc`. The block-specific implementations of `aFunc` are contained in the files `MyBlock.tlc` and `YourBlock.tlc`.

Using `GENERATE` calls, `polymorph.tlc` dispatches to the appropriate function for each block type. Notice that the `aFunc` implementations can refer to the fields of `MyBlock` and `YourBlock`, because these records are in generate scope.

- The following listing is `polymorph.tlc`.

```
%% polymorph.tlc

%language "C"

%%create records used as scopes within the dispatched functions

%createrecord MyRecord { Type "MyBlock"; data 123 }
%createrecord YourRecord { Type "YourBlock"; theStuff 666 }

%% dispatch the functions thru the GENERATE call.

%% dispatch to MyBlock implementation
%<GENERATE(MyRecord, "aFunc")>

%% dispatch to YourBlock implementation
%<GENERATE(YourRecord, "aFunc")>

%% end of polymorph.tlc
```

- The following listing is `MyBlock.tlc`.

```
%%MyBlock.tlc

%implements "MyBlock" "C"
```

```
%% aFunc is invoked thru a GENERATE call in polymorph.tlc.
%% MyRecord is in generate scope in this function.
%% Therefore, fields of MyRecord can be referenced without
%% qualification

%function aFunc(r) Output
%selectfile STDOUT
The value of MyRecord.data is: %<data>
%closefile STDOUT
%endfunction

%%end of MyBlock.tlc
```

- The following listing is `YourBlock.tlc`.

```
%%YourBlock.tlc

%implements "YourBlock" "C"

%% aFunc is invoked thru a GENERATE call in polymorph.tlc.
%% YourRecord is in generate scope in this function.
%% Therefore, fields of YourRecord can be referenced without
%% qualification

%function aFunc(r) Output
%selectfile STDOUT
The value of YourRecord.theStuff is: %<theStuff>
%closefile STDOUT
%endfunction

%%end of YourBlock.tlc
```

The invocation and output of `polymorph.tlc`, as displayed on the MATLAB console, are shown below:

```
tlc -v polymorph.tlc

The value of MyRecord.data is: 123
The value of YourRecord.theStuff is: 666
```

Note Functions defined in generate scope are local to that scope. This is an exception to the general rule that functions have global scope. In the above example, for instance, neither of the `aFunc` implementations has global scope.

The Scope Resolution Operator

The scope resolution operator (`::`) is used to indicate that the global scope should be searched when looking up a variable reference. The scope resolution operator is often used to change the value of global variables (or even create global variables) from within functions.

By using the scope resolution operator, you can resolve ambiguities that arise when a function references identically-named local and global variables. In the following example, a global variable `foo` is created. In addition, the function `myfunc` creates and initializes a local variable named `foo`. The function `myfunc` explicitly references the global variable `foo` by using the scope resolution operator.

```
%assign foo = 3   %% this variable has global scope
.
.
%function myfunc(arg)
   %assign foo = 3   %% this variable has local scope
   %assign ::foo = arg   %% this changes the global variable foo
%endfunction
```

You can also use the scope resolution operator, within a function, to create global variables. The following function creates and initializes a global variable:

```
%function sideffect(arg)
   %assign ::theglobal = arg   %% this creates a global variable
%endfunction
```

How TLC Resolves Variable References

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

Global Scope. In the simplest case, the Target Language Compiler resolves a variable reference by searching the global pool (including the `CompiledModel` structure).

%with Scope. You can modify the search list and search sequence by using the `%with` directive. For example, when you add the following construct

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

the `System[sysidx]` scope is added to the search list. This scope is searched first, as shown by this picture.

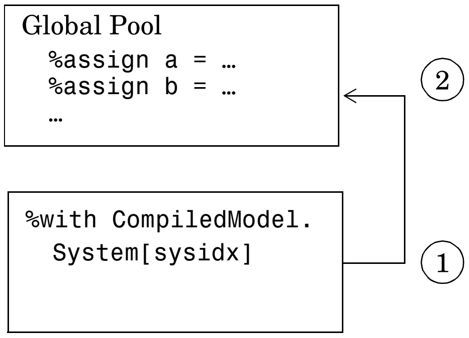


Figure 6-1: %with Scope Added to Search Sequence

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

```
Name
```

instead of

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

Function Scope. A function has its own scope. That scope is added to the previously described search list, as shown in this picture.

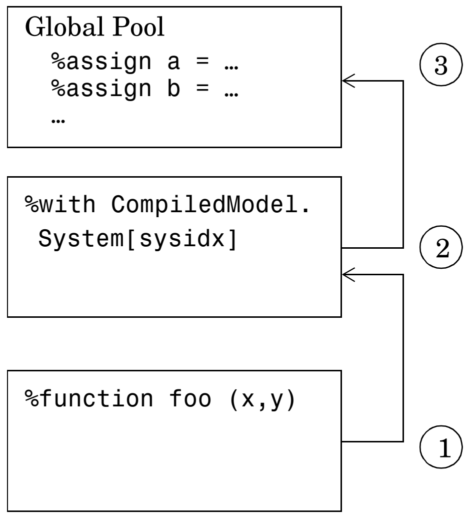


Figure 6-2: Scoping Rules Within Functions

For example, in the following code fragment:

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

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

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

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

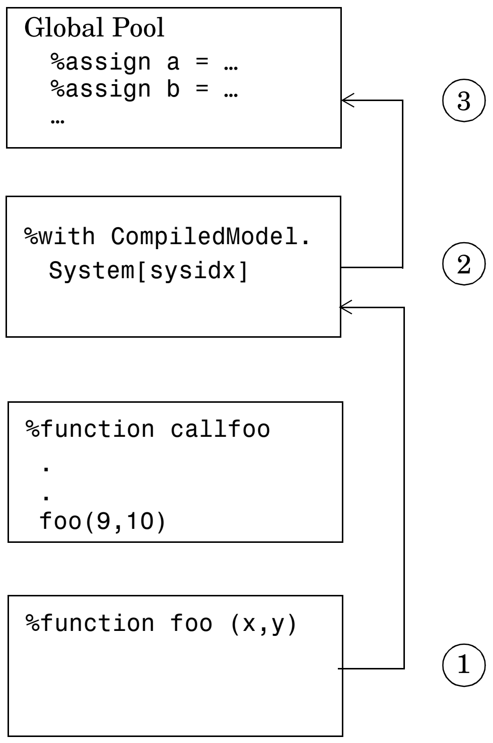


Figure 6-3: Nested File Scopes

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

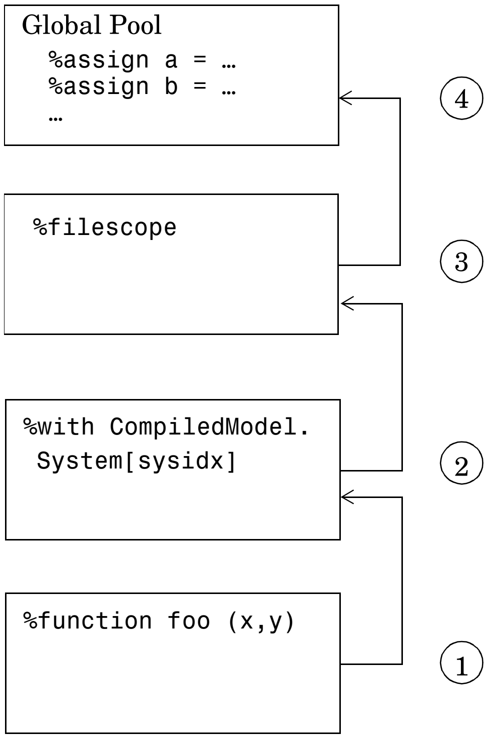


Figure 6-4: File Scopes Searched Before Global Scope

The rule for nested file scopes is similar to that for nested function scopes. In the case of nested file scopes, only the innermost nested file scope is searched.

Target Language Functions

The target language function construct is

```
%function identifier ( optional-arguments ) [Output | void]
%return
%endfunction
```

Functions in the target language are recursive and have their own local variable space. Target language functions do not produce any output, unless they explicitly use the `%openfile`, `%selectfile`, and `%closefile` directives, or are output functions.

A function optionally returns a value with the `%return` directive. The returned value can be any of the types defined in Table 6-2, Target Language Values.

In this example, a function, `name`, returns `x`, if `x` and `y` are equal, and returns `z`, if `x` and `y` are not equal:

```
%function name(x,y,z) void

%if x == y
   %return x
%else
   %return z
%endif

%endfunction
```

Function calls can appear in any context where variables are allowed.

All `%with` statements that are in effect when a function is called are available to the function. Calls to other functions do not include the local scope of the function, but do include any `%with` statements appearing within the function.

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

By default, a function returns a value and does not produce any output. You can override this behavior by specifying the `Output` and `void` modifiers on the function declaration line, as in

```
%function foo() Output
…
%endfunction
```

In this case, the function continues to produce output to the currently open file, if any, and is not required to return a value. You can use the `void` modifier to indicate that the function does not return a value, and should not produce any output, as in

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

Variable Scoping Within Functions

Within a function, the left-hand member of any `%assign` statement defaults to create a local variable. A new entry is created in the function's block within the scope chain; it does not affect any of the other entries. An example is shown in Figure 6-2.

You can override this default behavior by using `%assign` with the scope resolution operator `(::)`.

When you introduce new scopes within a function using `%with`, these new scopes are used during nested function calls, but the local scope for the function is not searched. Also, if a `%with` is included within a function, its associated scope is carried with any nested function call, as shown in Figure 6-5, Scoping Rules When Using %with Within a Function, on page 6-68.

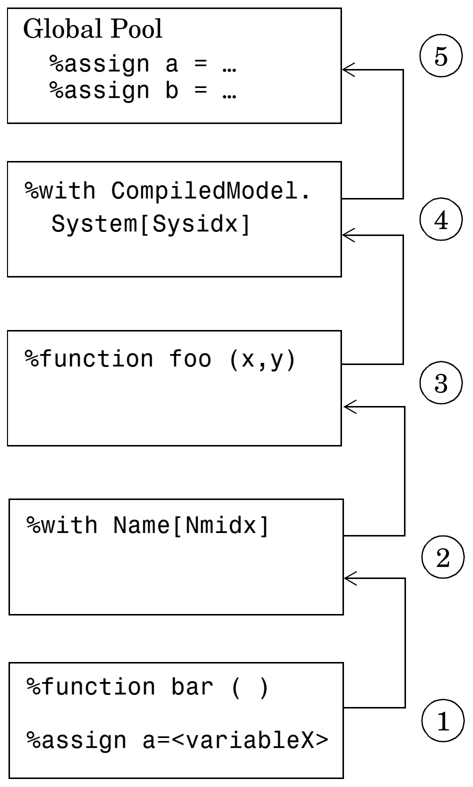


Figure 6-5: Scoping Rules When Using %with Within a Function

%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 6-8: Target Language Compiler Switches

Switch	Meaning
`-r filename`	Reads a database file (such as a `.rtw` file). Repeat this option multiple times to load multiple database files into the Target Language Compiler. Omit this option for target language programs that do not depend on the database.
`-v[number]`	Sets the internal verbosity level to `<number>`. Omitting this option sets the verbosity level to `1`.
`-Ipath`	Adds the specified directory to the list of paths to be searched for TLC files.
`-Opath`	Specifies that all output produced should be placed in the designated directory, including files opened with `%openfile` and `%closefile`, and `.log` files created in debug mode. To place files in the current directory, use `-O` (use the capital letter O, not zero).
`-m[number]`	Specifies the maximum number of errors to report is `<number>`. If no `-m` argument appears on the command line, it defaults to reporting the first five errors. If the `<number>` argument is omitted on this option, 1 is assumed.
`-xO`	Parse TLC file only (do not execute).
`-lint`	Performs some simple checks for performance and deprecated features.

Table 6-8: Target Language Compiler Switches (Continued)

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

As an example, the command line

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

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

Filenames and Search Paths

All target files have the `.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 `%addincludepath` directive, they will be searched after the current working directory, and in the order in which you specify them.

Debugging TLC Files

The Target Language Compiler debugger is a command line debugger that enables you to identify problems in executing TLC code. The following sections describe the facilities provided and provide examples of use. If you have not already done so, you may also wish to follow the tutorials “Debugging Your TLC Code” on page 3-40 and “Using TLC Code Coverage to Aid Debugging” on page 3-49.

About the TLC Debugger (p. 7-2)	Introducing the TLC debugging facility
Using the TLC Debugger (p. 7-3)	Enabling tracing and coverage, and command summary
TLC Coverage (p. 7-10)	Determining what TLC statements are executed
TLC Profiler (p. 7-13)	Measuring the execution time of each TLC function

About the TLC Debugger

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

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

The TLC debugger has a command line interface and uses commands similar to standard debugging tools such as `dbx` or `gdb`.

Tips for Debugging TLC Code

Here are a few tips that will help you to debug your TLC code:

1. To see the full TLC call stack, place the following statement in your TLC code before the line that is pointed to by the error message. This will be helpful in narrowing down your problem.

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

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

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

   Your message will appear when the Target Language Compiler is run with the `-v` command switch, but will be silent otherwise. You may use `%warning` instead of `%trace` to print variables, but you will need to remove or comment out such lines after you are through debugging.

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

Using the TLC Debugger

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

```
matlabroot/toolbox/rtw/rtwdemos/tlctutorial/tlcdebug/
```

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

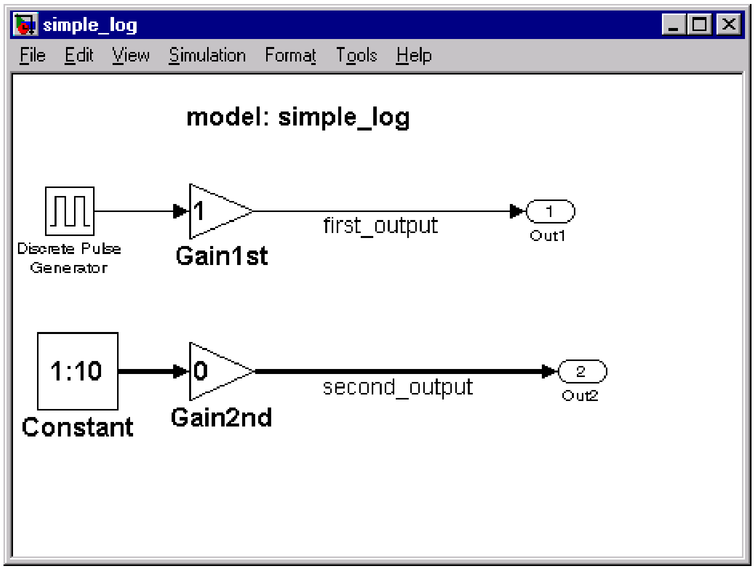
Note For TLC to use this local modified version of `gain.tlc`, which is not the one used by the actual Real-Time Workshop code-generation process, you must make sure the TLC file is in the same directory as the model.

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

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.

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 documentation.)

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.

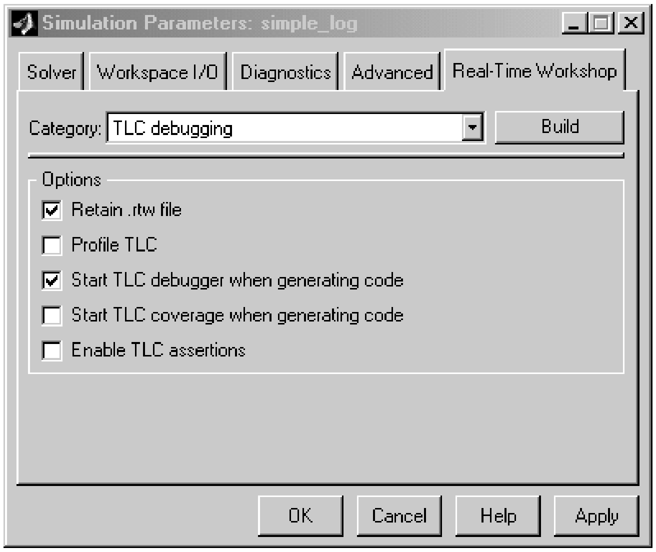
Invoke the Debugger

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

2 Select `TLC debugging` from the **Category** pull-down menu. From this screen you can select options that are specific to TLC debugging.

3 Select **Retain .rtw file**. This ensures that the `model.rtw` file is not deleted after code generation.

4 Select **Start TLC debugger when generating code** to invoke the TLC debugger when starting the code generation process. The dialog box should look like this.



Category: TLC debugging

Options

- [✓] Retain .rtw file
- [ ] Profile TLC
- [✓] Start TLC debugger when generating code
- [ ] Start TLC coverage when generating code
- [ ] Enable TLC assertions

Selecting **Start TLC debugger when generating code** is equivalent to adding `-dc` to the **System target file** field in the Real-Time Workshop pane of the **Simulation Parameters** dialog box.

5 Apply your changes and click **Build** to start code generation. This stops at the first line of executed TLC code, breaks into the TLC command line debugger, and displays the following prompt.

```
TLC_DEBUG>
```

You can now set breakpoints, explore the contents of Real-Time Workshop files, and explore variables in your TLC file using `print`, `which`, or `whos`.

An alternate way to invoke the TLC debugger is from the MATLAB prompt. (This assumes you retained the `model.rtw` file in the project directory.) To avoid making mistakes, we recommend copying the `tlc` command output by Real-Time Workshop to the MATLAB command window, and issue it after appending `-dc` to that command line.

A complete list of command line switches for the TLC debugger is found in Table 6-8, Target Language Compiler Switches, on page 6-70.

TLC Debugger Command Summary

Table 7-1, TLC Debugger Commands, on page 7-7 summarizes the TLC debugger commands.

To obtain more detailed help on individual commands, type

```
help command
```

from within the TLC debugger, as in this example:

```
TLC-DEBUG> help clear
```

You can abbreviate any TLC debugger command to its shortest unique form. For example,

```
TLC-DEBUG> break warning
```

can be abbreviated to

```
TLC-DEBUG> br warning
```

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

Table 7-1: TLC Debugger Commands

Command	Description
`assign variable=value`	Change a variable in the running program.
`break ["filename":]line|error|warning|trace|function`	Set a breakpoint. See also “%breakpoint Directive”
`clear [breakpoint#|all]`	Remove a breakpoint.
`condition [breakpoint#] [expression]`	Attach a condition to a breakpoint
`continue ["filename":]line|function`	Continue from a breakpoint.
`disable [breakpoint#]`	Disable a breakpoint.
`down [n]`	Move down the stack.
`enable [breakpoint#]`	Enable a breakpoint.
`finish`	Break after completing the current function
`help [command]`	Obtain help for a command.
`ignore [breakpoint#]count`	Set the ignore count of a breakpoint
`iostack`	Display contents of I/O stack
`list start[,end]`	List lines from the file from start to end.
`loadstate "filename"`	Load debugger breakpoint state from a file
`next`	Single step without going into functions.

Table 7-1: TLC Debugger Commands

Command	Description
`print expression`	Print the value of a TLC expression. To print a record, you must specify a fully qualified "scope" such as `CompiledModel.System[0].Block[0]`.
`quit`	Quit the TLC debugger. You can also exit the debugger by typing **Ctrl + C** at the prompt.
`run "filename"`	Run a batch file of command line debugger commands
`savestate "filename"`	Save debugger breakpoint state to a file
`status`	Display a list of active breakpoints.
`step`	Step into.
`stop ["filename":]line|error|warning|trace|function`	Set a breakpoint (same as `break`).
`tbreak ["filename":]line|function`	Set a temporary breakpoint
`thread [n]`	Change the active thread to thread #*n* (0 is the main program's thread number).
`threads`	List the currently active TLC execution threads.
`tstop ["filename":]line|function`	Set a temporary breakpoint
`up [`*n*`]`	Move up the stack.
`where`	Show the currently active execution chains.
`which name`	Look up the name and display what scope it comes from.
`whos [::|expression]`	List the variables in the given scope.

%breakpoint Directive

As an alternative to the `break` command, you can embed breakpoints at any point in a TLC file by adding the directive:

```
%breakpoint
```

Usage Notes

When using `break` or `stop`, use

- `error` to break or stop on error
- `warn` to break or stop on warning
- `trace` to break or stop on trace

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

```
TLC_DEBUG> break "gain.tlc":error
```

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:
[1] break File: gain.tlc Line: 46
[2] break File: gain.tlc Line: 25
TLC-DEBUG> clear 2
```

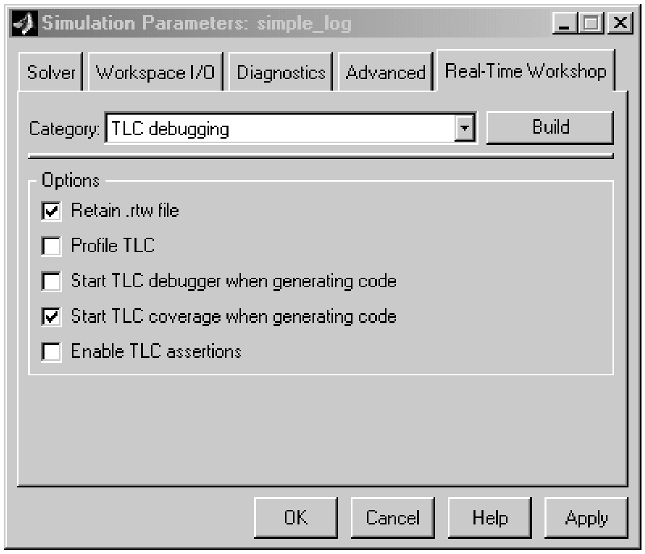
In this example, `clear 2` clears the second breakpoint.

TLC Coverage

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

Using the TLC Coverage Option

The TLC coverage option provides an easier way to ascertain that the different code parts (not paths) in your code are exercised. To initiate TLC coverage generation, select **Start TLC coverage when generating code** from the `TLC debugging` category of the **Real-Time Workshop** pane of the **Simulation Parameters** dialog box. You can also initiate TLC coverage from the command line switch `-dn` in the **System target file** field, but this is not recommended.



Category: TLC debugging

- [✓] Retain .rtw file
- [ ] Profile TLC
- [ ] Start TLC debugger when generating code
- [✓] Start TLC coverage when generating code
- [ ] Enable TLC assertions

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: %% $RCSfile: 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.

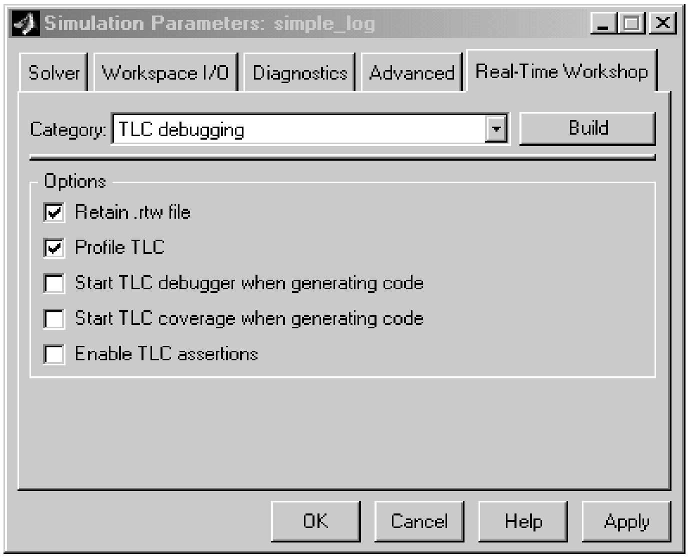
TLC Profiler

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** pane of the **Simulation Parameters** dialog box. Apply your changes and click the **Build** button.



Category: TLC debugging

- [✓] Retain .rtw file
- [✓] Profile TLC
- [ ] Start TLC debugger when generating code
- [ ] Start TLC coverage when generating code
- [ ] Enable TLC assertions

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.

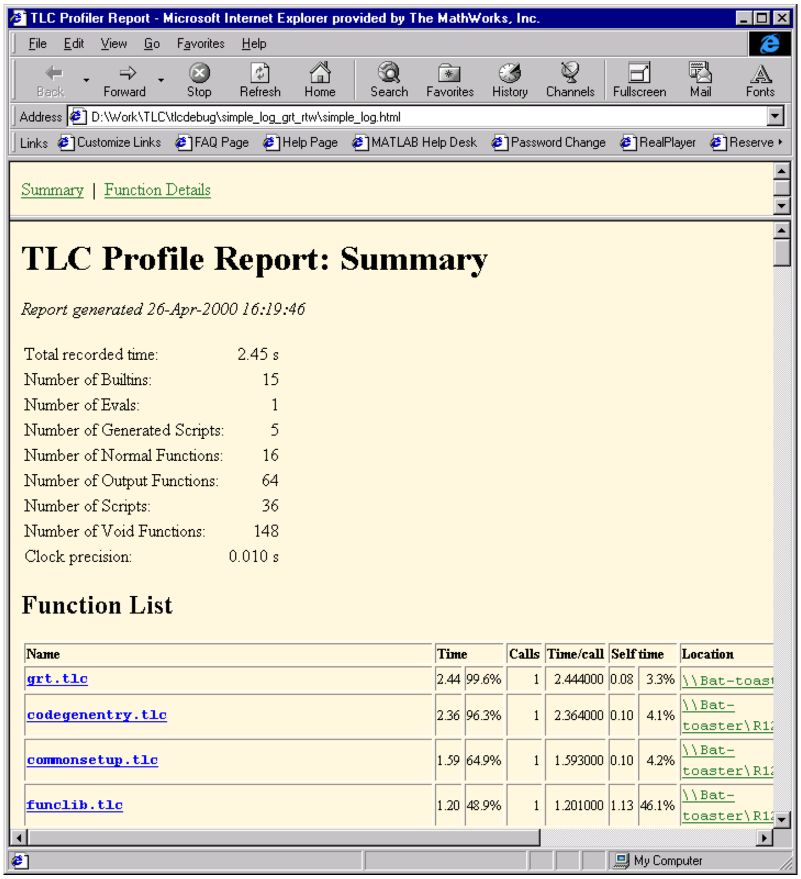
Analyzing the Report

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`.


TLC Profiler Report - Microsoft Internet Explorer provided by The MathWorks, Inc.
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History Channels Fullscreen Mail Fonts
Address D:\Work\TLC\tlcdebug\simple_log_grt_rtw\simple_log.html
Links Customize Links FAQ Page Help Page MATLAB Help Desk Password Change RealPlayer Reserve
Summary | Function Details
My Computer

TLC Profile Report: Summary

Report generated 26-Apr-2000 16:19:46

Total recorded time:	2.45 s
Number of Builtins:	15
Number of Evals:	1
Number of Generated Scripts:	5
Number of Normal Functions:	16
Number of Output Functions:	64
Number of Scripts:	36
Number of Void Functions:	148
Clock precision:	0.010 s

Function List

Name	Time		Calls	Time/call	Self time		Location
grt.tlc	2.44	99.6%	1	2.444000	0.08	3.3%	\\Bat-toas
codegenentry.tlc	2.36	96.3%	1	2.364000	0.10	4.1%	\\Bat-toaster\R1
commonsetup.tlc	1.59	64.9%	1	1.593000	0.10	4.2%	\\Bat-toaster\R1
funclib.tlc	1.20	48.9%	1	1.201000	1.13	46.1%	\\Bat-toaster\R1

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.

Nonexecutable Directives

TLC considers the following directives to be nonexecutable lines. Therefore, these directives are not counted in TLC Profiler reports:

- `%filescope`
- `%else`
- `%endif`

- `%endforeach`
- `%endfor`
- `%endroll`
- `%endwith`
- `%body`
- `%endbody`
- `%endfunction`
- `%endswitch`
- `%default`
- `any type of comment (%% or /% stuff %/)`

Improving Performance

Analyzing the profiler results also gives you an overview of which functions are used more often or are more expensive. Then, you can either improve those functions that were written by you, or try alternative methods to improve code generation speed. Two points to consider are

- Reduce usage of `EXISTS`. Performing an `EXISTS` on a field is more costly than comparing the field to a value. When possible, create an inert default value for a field. Then, instead of doing an `EXISTS` on the entity, compare it against the default value.
- Reduce the use of one line functions when they are not really needed. One line functions might be a bottleneck for code generation speed. When readability is not greatly impacted, consider expanding out the function.

Inlining S-Functions

To wrap or to inline, that is the question. Once you have decided, the following sections explain how to go about it, using the `timestwo` S-function as a running example. It turns out that inlining works almost identically for C, M-file and Fortran S-functions,

Writing Block Target Files to Inline S-Functions (p. 8-2)	Fully inlined versus wrapped S-functions
Inlining C MEX S-Functions (p. 8-4)	Calls made by C S-functions and how to handle them
Inlining M-File S-Functions (p. 8-17)	Accelerating M-file S-function performance
Inlining Fortran (F-MEX) S-Functions (p. 8-19)	The `timestwo` function coded in Fortran is handled the same
TLC Coding Conventions (p. 8-23)	Make your TLC code more robust by observing case conventions and using library functions
Block Target File Methods (p. 8-28)	Enumerates the functions needed to emit code for blocks
Loop Rolling (p. 8-36)	An example that handles multiple inputs
Error Reporting (p. 8-39)	Help in finding the source of trouble

Writing Block Target Files to Inline S-Functions

With C MEX S-functions, all targets except ERT will support calling the original C MEX code if the source code (`.c` file) is available when Real-Time Workshop enters its build phase. For S-functions that are in Fortran or `.m`, you must inline them in order to have complete code generation for Simulink models that contain them. Additionally, once you have determined that you will inline an S-function, you must decide to either make it *fully inlined* or *wrappered*.

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 Wrappered Code Generation

When the physical size of the code needed for a block becomes too large for inlining, the block target file is written to gather inputs, outputs, and parameters, and make a call to a function that you write to perform the block functionality. This has an advantage in generated code size when the code in the function is large or there are many instances of this block in a model. Of course, the overhead of the function call must be considered when weighing the option of fully inlining the block algorithm or generating function calls.

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

- Write all the function(s) once, put them in `.c` file(s) and have the TLC code's `BlockTypeSetup` method specify external references to your support functions. Use `LibAddToModelSources` for names of the modules containing the supporting functions. This approach is usually done using one function per file to get the smallest executable possible.
- Write a more sophisticated TLC file that in addition to the methods such as `Start` and `Outputs` will also conditionally generate more functions in separate code generation buffers to be written to a separate `.c` file that contains customized versions of functions (data types, widths, algorithms,

etc.), but only the functions needed by this model instead of all possible functions.

Either approach can produce optimal code. The first option can result in hundreds of files if your S-function supports many data types, signal widths and algorithm choices. The second approach is more difficult to write, but results in a more maintainable code generation library and the code can be every bit as tight as the first approach.

For further information on wrapping, see “Wrapper Inlined S-Function Example” on page 2-10, the tutorial “Wrapping User Code with TLC” on page 3-53, and Writing Wrapper S-Functions in Writing S-functions documentation.

Inlining C MEX S-Functions

When a Simulink model contains an S-function and a corresponding TLC block target file exists for that S-function, Real-Time Workshop inlines the S-function. Inlining an S-function can produce more efficient code by eliminating the S-function Application Program Interface (API) layer from the generated code.

For S-functions that can perform a variety of tasks, inlining them gives you the opportunity to generate code only for the current mode of operation set for each instance of the block. As an example of this, if an S-function accepts an arbitrary signal width and loops through each element of the signal, you would want to generate inlined code that has loops when the signal has two or more elements, but generates a simple nonlooped calculation when the signal has just one element.

Level 1 C MEX S-functions (written to an older form of the S-function API) that are not inlined will cause the generated code to make calls to all of these seven functions, even if the routine is empty for the particular S-function.

Function	**Purpose**
`mdlInitializeSizes`	Initialize the sizes array.
`mdlInitializeSampleTimes`	Initialize the sample times array.
`mdlInitializeConditions`	Initialize the states.
`mdlOutputs`	Compute the outputs.
`mdlUpdate`	Update discrete states.
`mdlDerivatives`	Compute the derivatives of continuous states.
`mdlTerminate`	Clean up when the simulation terminates.

Level 2 C MEX S-functions (i.e., those written to the current S-function API) that are not inlined make calls to the above functions with the following exceptions:

- `mdlInitializeConditions` is 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 `mdlRTW` method 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 `mdlRTW`	`Sizes` => `double`, `Prms` => `double` Error if non-C MEX S-function	`Prms` => `double` or string (`int16`)
Has `mdlRTW`	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 documentation.

Comparison of Noninlined and Inlined Versions of model.c

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

Noninlined S-Function.

```
/*
 * model.c
.
.
.
*/
real_T untitled_RGND = 0.0;             /* real_T ground */
/* Start the model */
void MdlStart(void)
{
  /* (no start code required) */
}
/* Compute block outputs */
void MdlOutputs(int_T tid)
{
  /* Level2 S-Function Block: <Root>/S-Function (foogain) */
  {
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnOutputs(rts, tid);
  }
}
/* Perform model update */
void MdlUpdate(int_T tid)
{
  /* (no update code required) */
}
/* Terminate function */
void MdlTerminate(void)
{
  /* Level2 S-Function Block: <Root>/S-Function (foogain) */
  {
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnTerminate(rts);
  }
}
#include "model_reg.h"
/* [EOF] model.c */
```

Inlined S-Function.

This code is *model*.c with the foogain S-function fully inlined:

```
/*
 * model.c
.
.
.
*/
/* Start the model */
void MdlStart(void)
{
  /* (no start code required) */
}

/* Compute block outputs */
void MdlOutputs(int_T tid)



  /* S-Function block: <Root>/S-Function */
  /* NOTE: There are no calls to the S-function API in the inlined
     version of model.c. */
  rtB.S_Function = 0.0 * rtP.S_Function_Gain;
}

/* Perform model update */
void MdlUpdate(int_T tid)
{
  /* (no update code required) */
}

/* Terminate function */
void MdlTerminate(void)
```

```
{
  /* (no terminate code required) */
}

#include "model_reg.h"

/* [EOF] model.c */
```

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

```
rtB.S_Function = 0.0 * rtP.S_Function_Gain;
```

Including a TLC file drastically decreased the code size and increased the execution efficiency of the generated code. These notes highlight some information about the TLC code and the generated output:

- The TLC directive `%implements` is required by all block target files, and must be the first executable statement in the block target file. This directive guarantees that the Target Language Compiler does not execute an inappropriate target file for S-function `foogain`.
- The input to `foo` is `rtGROUND` (a Real-Time Workshop global equal to 0.0) since `foo` is the only block in the model and its input is unconnected.
- Including a TLC file for `foogain` eliminated the need for an S-function registration segment for `foogain`. This significantly reduces code size.
- The TLC code will inline the `gain` parameter when Real-Time Workshop is configured to inline parameter values. For example, if the S-function parameter is specified as 2.5 in the S-function dialog box, the TLC `Outputs` function generates

  ```
  rtB.foo = input * 2.5;
  ```

- Use the `%generatefile` directive if your operating system has a filename size restriction and the name of the S-function is `foosfunction` (that exceeds the limit). In this case, you would include the following statement in the system target file (anywhere prior to a reference to this S-function's block target file).

```
%generatefile foosfunction "foosfunc.tlc"
```

This statement tells the Target Language Compiler to open `foosfunc.tlc` instead of `foosfunction.tlc`.

Comparison of Noninlined and Inlined Versions of model_reg.h

Inlining a Level 2 S-function significantly reduces the size of the *`model`*`_reg.h` code. Model registration functions are lengthy; much of the code has been eliminated in this example. The code below highlights the difference between the noninlined and inlined versions of `model_reg.h`; inlining eliminates all this code:

```
/*
 * model_reg.h
 *
.
.
.
*/
/* Normal model initialization code independent of
      S-functions  */

/* child S-Function registration */
  ssSetNumSFunctions(rtS, 1);

  /* register each child */
  {
    static SimStruct childSFunctions[1];
    static SimStruct *childSFunctionPtrs[1];

    (void)memset((char_T *)&childSFunctions[0], 0,
                  sizeof(childSFunctions));
    ssSetSFunctions(rtS, &childSFunctionPtrs[0]);
    {
      int_T i;

      for(i = 0; i < 1; i++) {
        ssSetSFunction(rtS, i, &childSFunctions[i]);
      }
    }
```

```
/* Level2 S-Function Block: untitled/<Root>/S-Function
   (foogain) */
{
  extern void foogain(SimStruct *rts);
  SimStruct *rts = ssGetSFunction(rtS, 0);

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

  {
    int_T i;

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

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

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

    /* port 0 */
    {
      static real_T const *sfcnUPtrs[1];

      sfcnUPtrs[0] = &untitled_RGND;
      ssSetInputPortWidth(rts, 0, 1);
      ssSetInputPortSignalPtrs(rts, 0,
          (InputPtrsType)&sfcnUPtrs[0]);
    }
  }

  /* outputs */
  {
    static struct _ssPortOutputs outputPortInfo[1];
    _ssSetNumOutputPorts(rts, 1);
    ssSetPortInfoForOutputs(rts, &outputPortInfo[0]);
    ssSetOutputPortWidth(rts, 0, 1);
    ssSetOutputPortSignal(rts, 0, &rtB.S_Function);
  }

  /* path info */
  ssSetModelName(rts, "S-Function");
  ssSetPath(rts, "untitled/S-Function");
  ssSetParentSS(rts, rtS);
  ssSetRootSS(rts, ssGetRootSS(rtS));
  ssSetVersion(rts, SIMSTRUCT_VERSION_LEVEL2);

  /* parameters */
  {
    static mxArray const *sfcnParams[1];

    ssSetSFcnParamsCount(rts, 1);
    ssSetSFcnParamsPtr(rts, &sfcnParams[0]);

    ssSetSFcnParam(rts, 0, &rtP.S_Function_P1Size[0]);
  }

  /* registration */
  foogain(rts);

  sfcnInitializeSizes(rts);
  sfcnInitializeSampleTimes(rts);

  /* adjust sample time */
  ssSetSampleTime(rts, 0, 0.2);
  ssSetOffsetTime(rts, 0, 0.0);
  sfcnTsMap[0] = 0;
```

```
        /* Update the InputPortReusable and BufferDstPort flags for
           each input port */
         ssSetInputPortReusable(rts, 0, 0);
         ssSetInputPortBufferDstPort(rts, 0, -1);

        /* Update the OutputPortReusable flag of each output port */
      }
    }
```

A TLC File to Inline S-Function foogain

To avoid unnecessary calls to the S-function and to generate the minimum code required for the S-function, the following TLC file, `foogain.tlc`, is provided as an example.

```
%implements "foogain" "C"

%function Outputs (block, system) Output
  /* %<Type> block: %<Name> */
  %%
  %assign y = LibBlockOutputSignal (0, "", "", 0)
  %assign u = LibBlockInputSignal (0, "", "", 0)
  %assign p = LibBlockParameter (Gain, "", "", 0)
  %<y> = %<u> * %<p>;

%endfunction
```

Managing Block Instance Data with an Eye Toward Code Generation

Instance data is extra data or working memory that is unique to each instance of a block in a Simulink model. This does not include parameter or state data (which is stored in the model parameter and state vectors, respectively), but rather is used for purposes such as caching intermediate results or derived representations of parameters and modes. One example of instance data is the buffer used by a transport delay block.

Allocating and using memory on an instance by instance basis can be done several ways in a Level 2 S-function: via `ssSetUserData`, work vectors (e.g., `ssSetRWork`, `ssSetIWork`), or data-typed work vectors known as `DWorks`. For the smallest effort in writing both the S-function and block target file and for automatic conformance to both static and malloc instance data on targets such as `grt` and `grt_malloc`, The MathWorks recommends using data-typed work vectors when writing S-functions with instance data, accessed with the `ssSetDWork` and `ssGetDWork` methods.

The advantages are twofold. In the first place, writing the S-function is more straightforward in that memory allocations and frees are handled for you by Simulink. Secondly, the `DWork` vectors are written to the `model.rtw` file for you automatically, including the `DWork` name, data type, and size. This makes writing the block target file a snap, since you have no TLC code to write for allocating and freeing the `DWork` memory — Real-Time Workshop takes care of this for you.

Additionally, if you want to bundle up groups of `DWorks` into structures for passing to functions, you can populate the structure with pointers to `DWork` arrays in both your S-function's `mdlStart` function and the block target file's `Start` method, achieving consistency between the S-function and the generated code's handling of data.

Finally, using `DWorks` makes it straightforward to create a specific version of code (data types, scalar vs. vectorized, etc.) for each block instance that matches the implementation in the S-function, i.e., both implementations use `DWorks` in the same way so that the inlined code can be used with the Simulink Accelerator without any changes to the C MEX S-function or the block target file.

Using Inlined Code With the Simulink Accelerator

By default, the Simulink Accelerator will call your C MEX S-function as part of an accelerated model simulation. If you want to instead have the accelerator inline your S-function before running the accelerated model, tell the accelerator to use your block target file to inline the S-function with the `SS_OPTION_USE_TLC_WITH_ACCELERATOR` flag in the call to `ssSetOptions()` in the `mdlInitializeSizes` function of that S-function.

Note that memory and work vector size and usage must be the same for the TLC generated code and the C MEX S-function, or the Simulink Accelerator will not be able to execute the inlined code properly. This is because the C MEX S-function is called to initialize the block and its work vectors, calling the `mdlInitializeSizes`, `mdlInitializeConditions`, `mdlCheckParameters`, `mdlProcessParameters`, and `mdlStart` functions. In the case of constant signal propagation, `mdlOutputs` is called from the C MEX S-function during the initialization phase of model execution.

During the time-stepping phase of accelerated model execution, the code generated by the `Output` and `Update` block TLC methods will execute, plus the `Derivatives` and zero-crossing methods if they exist. The `Start` method of the block target file are not used in generating code for an accelerated model.

Inlining M-File S-Functions

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

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

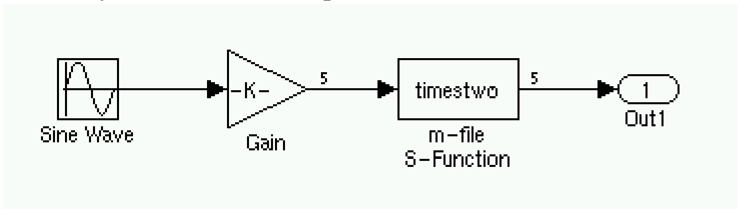
A quick example will illustrate the equivalence of C MEX and M-file S-functions for code generation. The M-file S-function `timestwo.m` is equivalent to the C MEX S-function `timestwo`. In fact, the TLC file for the C MEX S-function `timestwo` will work for the M-file S-function `timestwo.m` as well! Since TLC only requires the 'root' name of the S-function and not its type, it is independent of the type of S-function. In the case of `timestwo`, one line determines what the TLC file will be used for

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

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

```
which timestwo
```

The answer you see will be the M-file S-function `timestwo.m` in the temporary directory. Here is the sample model.



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

```
/* S-Function Block: <Root>/m-file S-Function */
  /* Multiply input by two */
  {
    int_T i1;
    const real_T *u0 = &rtB.Gain[0];
    real_T *y0 = &rtB.m_file_S_Function[0];

    for (i1=0; i1 < 5; i1++) {
      y0[i1] = u0[i1] * 2.0;
    }
  }
```

As expected, each of the inputs, `u0[i1]`, is multiplied by 2.0 to form the output value. The `Outputs` method in the block target file used to generate this code was

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

%endfunction
```

Alter these temporary copies of the M-file S-function and the TLC file to see how they interact — start out by just changing the comments in the TLC file and see it show up in the generated code, then work up to algorithmic changes.

Inlining Fortran (F-MEX) S-Functions

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

```
C
C     FTIMESTWO.FOR
C     $Revision: 1.1$
C
C     A sample FORTRAN representation of a
C     timestwo S-function.
C     Copyright 1990-2000 The MathWorks, Inc.
C
C=====================================================
C     Function:  SIZES
C
C     Abstract:
C       Set the size vector.
C
C       SIZES returns a vector which determines model
C       characteristics.  This vector contains the
C       sizes of the state vector and other
C       parameters. More precisely,
C       SIZE(1)  number of continuous states
C       SIZE(2)  number of discrete states
C       SIZE(3)  number of outputs
C       SIZE(4)  number of inputs
C       SIZE(5)  number of discontinuous roots in
C                the system
C       SIZE(6)  set to 1 if the system has direct
C                feedthrough of its inputs,
C                otherwise 0
C
C=====================================================
C
      SUBROUTINE SIZES(SIZE)
C     .. Array arguments ..
      INTEGER*4        SIZE(*)
```

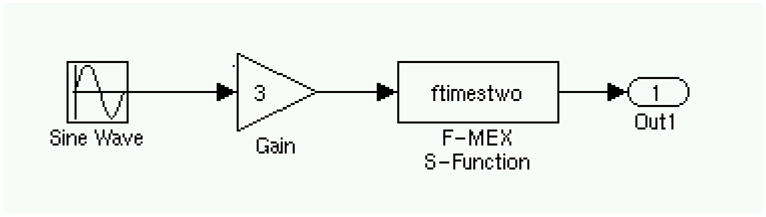
```
C     .. Parameters ..
      INTEGER*4       NSIZES
      PARAMETER       (NSIZES=6)

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

      RETURN
      END

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

      Y(1) = U(1) * 2.0

      RETURN
      END

C
C=====================================================
C
C     Stubs for unused functions.
C
C=====================================================
      SUBROUTINE INITCOND(XO)
      REAL*8          XO(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE DERIVS(T, X, U, DX)
      REAL*8          T, X(*), U(*), DX(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE DSTATES(T, X, U, XNEW)
      REAL*8          T, X(*), U(*), XNEW(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE DOUTPUT(T, X, U, Y)
      REAL*8          T, X(*), U(*), Y(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE TSAMPL(T, X, U, TS, OFFSET)
      REAL*8          T,TS,OFFSET,X(*),U(*)
C --- Nothing to do.
      RETURN
      END

      SUBROUTINE SINGUL(T, X, U, SING)
      REAL*8          T, X(*), U(*), SING(*)
C --- Nothing to do.
      RETURN
      END
```

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

Putting this into an S-function block in a simple model will illustrate the interface for inlining the S-function. Once your Fortran MEX environment is set up, prepare the code for use by compiling the S-function in a working directory along with the file `simulink.for` from *`matlabroot`*`/simulink/src/`. This is done with the `mex` command at the MATLAB command prompt:

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

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



The TLC code needed to inline this block is a modified form of the now familiar `timestwo.tlc`. In your working directory, create a file named `ftimestwo.tlc` and put this code into it.

```
%implements "ftimestwo" "C"

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

Now you can generate code for the `ftimestwo` Fortran MEX S-function. The resulting code fragment specific to `ftimestwo` is

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

TLC Coding Conventions

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

Begin Identifiers with Uppercase Letters

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

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

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

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

Since the `Name` identifier within the `RWorkDefine` record is promoted to `PrevT` in its parent scope, it must start with an uppercase letter. The promotion of the `Name` identifier into the parent block scope is currently done for the `Parameter`, `RWorkDefine`, `IWorkDefine`, and `PWorkDefine` block records.

The Target Language Compiler assignment directive (`%assign`) generates a warning if you assign a value to an “unqualified” Real-Time Workshop identifier. For example,

```
%assign TID = 1
```

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

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

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

Begin Global Variable Assignments with Uppercase Letters

Global TLC variable assignments should start with uppercase letters. A global variable is any variable declared in a system target file (grt.tlc, mdlwide.tlc, mdlhdr.tlc, mdlbody.tlc, mdlreg.tlc, or mdlparam.tlc), or within a function that uses the :: operator. In some sense, global assignments have the same scope as Real-Time Workshop variables. An example of a global TLC variable defined in mdlwide.tlc is

```
%assign InlineParameters = 1
```

An example of a global reference in a function is

```
%function foo() void
  %assign ::GlobalIdx = ::GlobalIdx + 1
%endfunction
```

Begin Local Variable Assignments with Lowercase Letters

Local TLC variable assignments should start with lowercase letters. A local TLC variable is a variable assigned inside a function. For example,

```
%assign numBlockStates = ContStates[0]
```

Begin Functions Declared in block.tlc files with Fcn

When you declare a function inside a block.tlc file, it should start with Fcn. For example:

```
%function FcnMyBlockFunc(...)
```

Note Functions declared inside a system file are global; functions declared inside a block file are local.

Do Not Hard Code Variables Defined in commonsetup.tlc

Since the Real-Time Workshop tracks use of variables and generates code based on usage, you should use access routines instead of directly using a variable. For example, you should not use the following in your TLC file:

```
x = %<tInf>;
```

You should use

```
x = %<LibRealNonFinite(inf)>;
```

Similarly, instead of using `%<tTID>`, use `%<LibTID()>`. For a complete list of functions, see 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:
Note c          { %%      Y = U * K
                { %%
                { %function Outputs(block, system) Output
                   /* %<Type> Block: %<Name> */  ────────} Note a
                   %assign rollVars = ["U", "Y", "P"]  ────────────────} Note e
Notes d, f      {  %roll sigIdx = RollRegions, lcv = RollThreshold, block,...
                {        "Roller", rollVars
                    %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
                    %assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
                    %assign k = LibBlockParameter(Gain, "", lcv, sigIdx)
                    %<y> = %<u> * %<k>;
                   %endroll
                                  ────────} Note b
                  %endfunction
```

Notes about this TLC code:

- **a** The code section for each block begins with a comment specifying the block type and name.
- **b** Include a blank line immediately after the end of the function in order to create consistent spacing between blocks in the output code.
- **c** Try to stay within 80 columns per line for the function banner. You might set up an 80 column comment line at the top of each function. As an example, see `constant.tlc`.
- **d** For consistency, use the variables `sysIdx` and `blkIdx` for system index and block index, respectively.
- **e** Use the variable `rollVars` when using the `%roll` construct.
- **f** When naming loop control variables, use `sigIdx` and `lcv` when looping over `RollRegions` and `xidx` and `xlcv` when looping over the states.

Example: `Output` function in `gain.tlc`

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

Example: `InitializeConditions` function in `linblock.tlc`

```
%roll xidx = [0:nStates-1], xlcv = RollThreshold,...
      block, "Roller", rollVars
```

Conditional Inclusion in Library Files

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

The convention is to use a variable with the same name as the base filename, uppercased and with underscores attached at both ends. So, a file named `customlib.tlc` should have the variable `_CUSTOMLIB_` guarding it.

As an example, the main Target Language Compiler function library, `funclib.tlc`, contains this TLC code to prevent multiple inclusion:

```
%if EXISTS("_FUNCLIB_") == 0
```

```
%assign _FUNCLIB_ = 1
  .
  .
  .
%endif  %% _FUNCLIB_
```

Block Target File Methods

Each block has a target file that determines what code should be generated for the block. The code can vary depending on the exact parameters of the block or the types of connections to it (e.g., wide vs. scalar input).

Within each block target file, *block functions* specify the code to be output for the block in the model's or subsystem's `start` function, `output` function, `update` function, and so on.

Block Target File Mapping

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

Block Functions

The functions declared inside each of the block target files are called by the system target files. In these tables, `block` refers to a Simulink block name (e.g., `gain` for the Gain block) and `system` refers to the subsystem in which the block resides. The first table lists the two functions that are used for preprocessing and setup. Neither of these functions outputs any generated code.

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

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

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

In object-oriented programming terms, these functions are polymorphic in nature since each block target file contains the same functions. The Target Language Compiler dynamically determines at run-time which block function to execute depending on the block's type. That is, the system file only specifies that the `Outputs` function, for example, is to be executed. The particular `Outputs` function is determined by the Target Language Compiler depending on the block's type.

To write a block target file, use these polymorphic block functions combined with the Target Language Compiler library functions. For a complete list of the Target Language Compiler library functions, see Chapter 9, "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/tlc/blocks/lookup2d.tlc` for an example of the `BlockInstanceSetup` function.

Syntax. `BlockInstanceSetup(block, system) void`

`block` = Reference to a Simulink block

`system` = Reference to a nonvirtual Simulink subsystem

This example uses `BlockInstanceSetup`:

```
%function BlockInstanceSetup(block, system) void
%if (block.InMask == "yes")
    %assign blockName = LibParentMaskBlockName(block)
  %else
    %assign blockName = LibGetFormattedBlockPath(block)
  %endif
  %if (CodeFormat == “Embedded-C”)
    %if !(ParamSettings.ColZeroTechnique == "NormalInterp" && ...
          ParamSettings.RowZeroTechnique == "NormalInterp")
      %selectfile STDOUT
```

```
Note: Removing repeated zero values from the X and Y axes will
produce more efficient code for block: %<blockName>.  To locate
this block, type

open_system('%<blockName>')

at the MATLAB command prompt.

      %selectfile NULL_FILE
    %endif
  %endif

%endfunction
```

BlockTypeSetup(block, system)

`BlockTypeSetup` executes once per block type before code generation begins. That is, if there are 10 Lookup Table blocks in the model, the `BlockTypeSetup` function in `look_up.tlc` is only called one time. Use this function to perform general work for all blocks of a given type.

See "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.

Enable(block, system)

Nonvirtual subsystem `Enable` functions are created whenever a Simulink subsystem contains a block with an `Enable` function. Including the `Enable` function in a block's target file places the block's specific enable code into this subsystem `Enable` function. See `sin_wave.tlc` for an example of the `Enable` function.

```
  %% Function: Enable ============================================
  %% Abstract:
  %% Subsystem Enable code is only required for the discrete form
  %% of the Sine Block. Setting the boolean to TRUE causes the
  %% Output function to re-sync its last values of cos(wt) and
  %% sin(wt).
  %%
  %function Enable(block, system) Output
    %if LibIsDiscrete(TID)
      /* %<Type> Block: %<Name> */
      %<LibBlockIWork(SystemEnable, "", "", 0)> = (int_T) TRUE;

    %endif
  %endfunction
```

Disable(block, system)

Nonvirtual subsystem `Disable` functions are created whenever a Simulink subsystem contains a block with a `Disable` function. Including the `Disable` function in a block's target file places the block's specific disable code into this subsystem `Disable` function. See `outport.tlc` in *`matlabroot`*`/rtw/c/tlc/blocks` for an example of the `Disable` function.

Start(block, system)

Include a `Start` function to place code into the `Start` function. The code inside the `Start` function executes once and only once. Typically, you include a `Start` function to execute code once at the beginning of the simulation (e.g., initialize values in the work vectors; see `backlash.tlc`) or code that does not need to be reexecuted when the subsystem in which it resides enables. See `constant.tlc` for an example of the `Start` function:

```
%% Function: Start ============================================
%% Abstract:
%% Set the output to the constant parameter value if the block
%% output is visible in the model’s start function scope, i.e.,
%% it is in the global rtB structure.
%%
%function Start(block, system) Output
  %if  LibBlockOutputSignalIsInBlockIO(0)
    /* %<Type> Block: %<Name> */
    %assign rollVars = [“Y”, “P”]
    %roll idx = RollRegions, lcv = RollThreshold, block, ...
      “Roller”, rollVars
      %assign yr = LibBlockOutputSignal(0,””, lcv, ...
        “%<tRealPart>%<idx>”)
      %assign pr = LibBlockParameter(Value, ““, lcv, ...
        “%<tRealPart>%<idx>”)
      %<yr> = %<pr>;
      %if LibBlockOutputSignalIsComplex(0)
        %assign yi = LibBlockOutputSignal(0, ““, lcv, ...
          “%<tImagPart>%<idx>”)
        %assign pi = LibBlockParameter(Value, ““, lcv, ...
          “%<tImagPart>%<idx>”)
        %<yi> = %<pi>;
      %endif
    %endroll

  %endif
%endfunction %% Start
```

InitializeConditions(block, system)

TLC code that is generated from the block's InitializeConditions function ends up in one of two places. A nonvirtual subsystem contains an Initialize function when it is configured to reset states on enable. In this case, the TLC code generated by this block function is placed in the subsystem Initialize function and the start function will call this subsystem Initialize function. If, however, the Simulink block resides in the root system or in a nonvirtual subsystem that does not require an Initialize function, the code generated from this block function is placed directly (inlined) into the start function.

There is a subtle difference between the block functions Start and InitializeConditions. Typically, you include a Start function to execute code that does not need to re-execute when the subsystem in which it resides enables. You include an InitializeConditions function to execute code that must reexecute when the subsystem in which it resides enables. See delay.tlc for an example of the InitializeConditions function. The following code is an example from ratelim.tlc:

```
%% Function: InitializeConditions =============================
%%
%% Abstract:
%% Invalidate the stored output and input in rwork[1 ...
%% 2*blockWidth] by setting the time stamp (stored in
%% rwork[0]) to rtInf.
%%
%function InitializeConditions(block, system) Output
  /* %<Type> Block: %<Name> */
  %<LibBlockRWork(PrevT, "", "", 0)> = %<LibRealNonFinite(inf)>;

%endfunction %% InitializeConditions
```

Outputs(block, system)

A block should generally include an Outputs function. The TLC code generated by a block's Outputs function is placed in one of two places. The code is placed directly in the model's Outputs function if the block does not reside in a nonvirtual subsystem and in a subsystem's Outputs function if the block resides in a nonvirtual subsystem. See absval.tlc for an example of the Outputs function:

```
%% Function: Outputs ==========================================
%% Abstract:
%%      Y[i] = fabs(U[i]) if U[i] is real or
%%      Y[i] = sqrt(U[i].re^2 + U[i].im^2) if U[i] is complex.
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %%
  %assign inputIsComplex = LibBlockInputSignalIsComplex(0)
  %assign RT_SQUARE = “RT_SQUARE”
  %%
  %assign rollVars = [“U”, “Y”]
  %if inputIsComplex
    %roll sigIdx = RollRegions, lcv = RollThreshold, ...
      block, “Roller”, rollVars
      %%
      %assign ur = LibBlockInputSignal( 0, ““, lcv, ...
        “%<tRealPart>%<sigIdx>”)
      %assign ui = LibBlockInputSignal( 0, ““, lcv, ...
        “%<tImagPart>%<sigIdx>”)
      %%
      %assign y = LibBlockOutputSignal(0, ““, lcv, sigIdx)
     %<y> = sqrt( %<RT_SQUARE>( %<ur> ) + %<RT_SQUARE>( %<ui> ) );
    %endroll
  %else
    %roll sigIdx = RollRegions, lcv = RollThreshold, ...
      block, “Roller”, rollVars
      %assign u = LibBlockInputSignal (0, ““, lcv, sigIdx)
      %assign y = LibBlockOutputSignal(0, ““, lcv, sigIdx)
      %<y> = fabs(%<u>);
    %endroll
  %endif

%endfunction
```

Note Zero-crossing reset code is placed in the `Outputs` function.

Update(block, system)

Include an `Update` function if the block has code that needs to be updated at each major time step. Code generated from this function is either placed into the model's or the subsystem's `Update` function, depending on whether or not the block resides in a nonvirtual subsystem. See `delay.tlc` for an example of the `Update` function.

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

%endfunction %% Update
```

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

Derivatives(block, system)

Include a `Derivatives` function when generating code to compute the block's continuous states. Code generated from this function is either placed into the model's or the subsystem's `Derivatives` function, depending on whether or not the block resides in a nonvirtual subsystem. See `integrat.tlc` for an example of the `Derivatives` function.

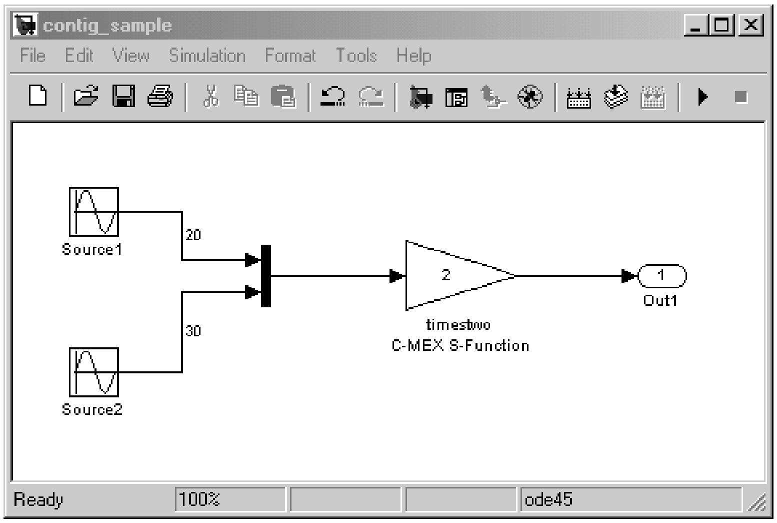
Terminate(block, system)

Include a `Terminate` function to place any code into `MdlTerminate`. User-defined S-function target files can use this function to save data, free memory, reset hardware on the target, and so on. See `tofile.tlc` for an example of the `Terminate` function.

Loop Rolling

One of the optimization features of the Target Language Compiler is the intrinsic support for loop rolling. Based on a specified threshold, code generation for looping operations can be unrolled or left as a loop (rolled).

Coupled with loop rolling is the concept of noncontiguous signals. Consider the following model.



The input to the `timestwo` S-function comes from two arrays located at two different memory locations, one for the output of `source1` and one for the output of block `source2`. This is because of a Simulink optimization feature that makes the mux block *virtual*, meaning that there is no code explicitly generated for the mux and thus no processor cycles spent evaluating it (i.e., it becomes a pure graphical convenience for the block diagram). So this is represented in the `model.rtw` file in this case as

```
Block {
  .
  .
  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.source2[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

```
ssSetInputPortRequiredContiguous(S, 1);
```

in the `mdlInitializeSizes` function to cause Simulink to implicitly generate code that performs a buffering operation. This option uses both extra memory and CPU cycles at runtime, but may be worth it if your algorithm performance increases enough to offset the overhead of the buffering.

This is accomplished by using the `%roll` directive. There is a tutorial covering the `%roll` directive in Chapter 3, "More on TLC Loop Rolling." See also `%roll`

`%endroll` on page 6-11 for the reference entry for `%roll` and `%roll` on page 6-31 for a section describing the behavior of `%roll`.

Error Reporting

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

To report an error condition detected in you TLC code, use the `LibBlockReportError` or `LibBlockReportFatalError` utility functions. Use of these functions is fully documented in the Reference section. Here is an example of using `LibBlockReportError` in the paramlib.tlc function `LibBlockParameter`: to report the condition of an improper use of that function:

```
%if TYPE(param.Value) == "Matrix"
    %% exit if the parameter is a true matrix,
    %% i.e., has more than one row or columns.
    %if nRows > 1
      %assign errTxt = "Must access parameter %<param.Name> using "...
        "LibBlockMatrixParameter."
      %<LibBlockReportError([], errTxt)>
    %endif
  %endif
```

Browse through *`matlabroot`*`/rtw/c/tlc` for more examples of the use of `LibBlockReportError`. Also, read further details in Appendix B, “TLC Error Handling,” which describes types of TLC errors and their interpretations.

TLC Function Library Reference

This chapter provides a set of Target Language Compiler functions that are useful for inlining S-functions. The TLC files contain many other library functions, but you should use only the functions that are documented in these reference pages for development. Undocumented functions may change significantly from release to release. A table of obsolete functions and their replacements is shown in Obsolete Functions.

Obsolete Functions (p. 9-2)	Deprecated functions and their replacements
Target Language Compiler Functions (p. 9-4)	Function syntax, conventions, and common arguments
Input Signal Functions (p. 9-9)	Functions that process and report on input signals
Output Signal Functions (p. 9-19)	Functions that process and report on output signals
Parameter Functions (p. 9-23)	Functions that process model parameters
Block State and Work Vector Functions (p. 9-28)	Functions that handle storage and states
Block Path and Error Reporting Functions (p. 9-31)	Functions for navigating paths and handling error conditions
Code Configuration Functions (p. 9-33)	Functions for tailoring code elements and comments
Sample Time Functions (p. 9-37)	Functions for handling continuous and discrete time
Other Useful Functions (p. 9-43)	Functions not elsewhere classified
Advanced Functions (p. 9-47)	Functions generally required only for special situations

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

Obsolete Functions

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

Obsolete Function	Equivalent Replacement Function
`LibBlockOutportLocation`	`LibBlockDstSignalLocation`
`LibCacheGlobalPrmData`	Use the block function `Start`
`LibContinuousState`	`LibBlockContinuousState`
`LibControlPortInputSignal`	`LibBlockSrcSignalLocation`
`LibDataInputPortWidth`	`LibBlockInputSignalWidth`
`LibDataOutputPortWidth`	`LibBlockOutputSignalWidth`

`LibDefineIWork`
`LibDefinePWork`
`LibDefineRWork`
	`IWork` , `PWork`, and `RWork` names are now specified via the `mdlRTW` function in your `C-MEX` S-function.

`LibDiscreteState`	`LibBlockDiscreteState`
`LibExternalResetSignal`	`LibBlockInputSignal`
`LibMapSignalSource`	`FcnMapDataTypedSignalSource`
`LibMaxBlockIOWidth`	Function is not used in the Real-Time Workshop.
`LibMaxDataInputPortWidth`	Function is not used in the Real-Time Workshop.
`LibMaxDataOutputPortWidth`	Function is not used in the Real-Time Workshop.
`LibPathName`	`LibGetBlockPath`, `LibGetFormattedBlockPath`
`LibPrevZCState`	`LibBlockPrevZCState`

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

Target Language Compiler Functions

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

Common Function Arguments

Several functions take similar or identical arguments. To simplify the reference pages, some of these arguments are documented in detail here instead of in the reference pages.

Argument	Description
`portIdx`	Refers to an input or output port index, starting at zero. For example the first input port of an S-function is 0.
`ucv`	User control variable. This is an advanced feature that overrides the `lcv` and `sigIdx` parameters. When used within an inlined S-function, it should generally be specified as `""`.
`lcv`	Loop control variable. This is generally generated by the `%roll` directive via the second `%roll` argument (e.g., `lcv=RollThreshold`) and should be passed directly to the library function. It will contain either `""`, indicating that the current pass through the `%roll` is being inlined, or it will be the name of a loop control variable such as `"i"`, indicating that the current pass through the `%roll` is being placed in a loop. Outside of the `%roll` directive, this is usually specified as `""`.

Argument	Description

`sigIdx`
or
`idx`
	Signal index. Sometimes referred to as the signal element index. When accessing specific elements of an input or output signal directly, the call to the various library routines should have `ucv=""`, `lcv=""`, and `sigIdx` equal to the desired integer signal index starting at 0. Note, for complex signals, `sigIdx` can be an overloaded integer index specifying both whether the real or imaginary part is being accessed and which element. When accessing these items inside of a `%roll`, the `sigIdx` generated by the `%roll` directive should be used.
	
	Most functions that take a `sigIdx` argument accept it in an overloaded form where `sigIdx` can be:
	
	• An integer, e.g. `3.` If the referenced signal is complex, then this refers to the identifier for the complex container. If the referenced signal is not complex, then this refers to the identifier.
	• An *id-num* usually of the form (see “Overloading sigIdx” on page 9-6):
	
	a `"%<tRealPart>%<idx>"` (e.g., `"re3"`). The real part of the signal element. Usually `"%<tRealPart>%<sigIdx>"` when `sigIdx` is generated by the `%roll` directive.
	
	b `"%<tImagPart>%<idx>"` (e.g., `"im3"`). The imaginary part of the signal element or `""` if the signal is not complex. Usually `"%<tImagPart>%<sigIdx>"` when `sigIdx` is generated by the `%roll` directive.
	
	The `idx` name is used when referring to a state or work vector.
	
	Functions that accept the three arguments `ucv`, `lcv`, `sigIdx` (or `idx`) are called differently depending upon whether or not they are used with in a `%roll` directive. If they are used within a `%roll` directive, ucv is generally specified as `""` and `lcv` and `sigIdx` are the same as those specified in the `%roll` directive. If they are not used with in a `%roll` directive, `ucv` and `lcv` are generally specified as `""` and `sigIdx` specifies which index to access.

Argument	Description
`paramIdx`	Parameter index. Sometimes referred to as the parameter element index. The handling of this parameter is very similar to `sigIdx` (i.e., it can be `#`, `re#`, or `im#`).
`stateIdx`	State index. Sometimes referred to as the state vector element index. It must evaluate to an integer where the first element starts at `0`.

Overloading sigIdx

The signal index (`sigIdx` sometimes written as `idx`) can be overloaded when passed to most library functions. Suppose we are interested in element 3 of a signal, and `ucv=""`, `lcv=""`. The following table shows:

- Values of `sigIdx`
- Whether the signal being referenced is complex
- What the function that uses `sigIdx` returns
- An example of a returned variable
- Data type of the returned variable

Note that "container" in the following table refers to the object that encapsulates both the real and imaginary parts of the number, e.g., `creal_T` defined in *`matlabroot`*`/extern/include/tmwtypes.h`.

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

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 `idx`, whether the signal is complex, and what the function that uses `idx` returns.

sigIdx	Complex	Function Returns
`"re3"`	yes	Real part of element i
`"im3"`	yes	Imaginary part of element i
`"3"`	yes	Complex container of element i
`3`	yes	Complex container of element i
`"re3"`	no	Element i
`"im3"`	no	`""`
`"3"`	no	Element i
`3`	no	Element i

Notes

- The vector index is only added for wide signals.
- If `ucv` is not an empty string (" "), then the `ucv` is used instead of `sigIdx` in the above examples and both `lcv` and `sigIdx` are ignored.

- If `ucv` is empty but `lcv` is not empty, then this function returns `"&y%<portIdx>[%<lcv>]"` and `sigIdx` is ignored.
- It is assumed here that the roller has appropriately declared and initialized the variables accessed inside the roller. The variables accessed inside the roller should be specified using `"rollVars"` as the argument to the `%roll` directive.

Input Signal Functions

LibBlockInputSignal(portIdx, ucv, lcv, sigIdx)

Based on the input port number (`portIdx`), the user control variable (`ucv`), the loop control variable (`lcv`), the signal index (`sigIdx`), and where this input signal is coming from, `LibBlockInputSignal` returns the appropriate reference to a block input signal.

The returned string value is a valid `rvalue` (right-side value) for an expression. The block input signal can come from another block, a state vector, an external input, or it can be a literal constant (e.g, 5.0).

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

Since the returned value can be a literal constant, you should not use `LibBlockInputSignal` to access the address of an input signal. To access the address of an input signal, use `LibBlockInputSignalAddr`. Accessing the address of the signal via `LibBlockInputSignal` may result in a reference to a literal constant (e.g., 5.0).

For example, the following would *not* work.

```
%assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
x = &%<u>;
```

If `%<u>` refers to an invariant signal with a value of `4.95`, the statement (after being processed by the pre-processor) would be generated as

```
x = &4.95;
```

or, if the input signal sources to ground, the statement could come out as

```
x = &0.0;
```

neither of these would compile.

Avoid any such situations by using `LibBlockInputSignalAddr`.

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

Real-Time Workshop tracks signals and parameters accessed by their address and declares them in addressable memory.

Input Arguments

The following table summarizes the input arguments to `LibBlockInputSignal`.

Table 9-1: LibBlockInputSignal Arguments

Argument	Description
`portIdx`	Integer specifying the input port index (zero-based). Note: for certain built-in blocks, `portIdx` can be a string identifying the port (such as `"enable"` or `"trigger"`).
`ucv`	User control variable. Must be a string, either an indexing expression or "".
`lcv`	Loop control variable. Must be a string, either an indexing expression or "".

`sigIdx`	Either an integer literal or a string of the form
	
	`%<tRealPart>Integer`
	`%<tImagPart>Integer`
	
	For example, the following signifies the real part of the signal and the imaginary part of the signal starting at 5:
	
	`"%<tRealPart>5"`
	`"%<tImagPart>5"`

General Usage

Uses of `LibBlockInputSignal` fall into the categories described below.

Direct indexing. If `ucv == ""` and `lcv == ""`, `LibBlockInputSignal` returns an indexing expression for the element specified by `sigIdx`.

Loop rolling/unrolling. In this case, `lcv` and `sigIdx` are generated by the `%roll` directive, and `ucv` must be `""`. A non-empty value for `lcv` is only allowed when generated by the `%roll` directive and when using the Roller TLC file (or a user supplied Roller TLC file that conforms to the same variable/signal offset handling). In addition, calls to `LibBlockInputSignal` with `lcv` should occur only when `"U"` or a specific input port (e.g. `"u0"`) is passed to the `%roll` directive via the roll variables argument.

The following example is appropriate for a single input/single output port S-function.

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

With the `%roll` directive, `sigIdx` is always the starting index of the current roll region and `lcv` will be `""` or an indexing variable. The following are examples of valid values:

- Example 1:

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

- Example 2:

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

In Example 1, `LibBlockInputSignal` returns `rtB.blockname[2]` when the input port is connected to the output of another block and:

- The loop control variable (lcv) generated by the %roll directive is empty, indicating that the current roll region is below the roll threshold and sigIdx is 0.
- The width of the input port is 1, indicating that this port is being scalar expanded.

  If sigIdx was non-zero, then rtB.blockname[sigIdx] would be returned. For example if sigIdx was 3, then rtB.blockname[3] would be returned.

In Example 2, LibBlockInputSignal returns u[i] when the current roll region is above the roll threshold and the input port width is non-scalar (wide). In this case, the Roller TLC file sets up a local variable, u, to point to the input signal and the code in the current %roll directive is placed within a for loop.

For another example, suppose we have a block with multiple input ports where each port has a width greater than or equal to 1 and at least one port has width equal to 1. The following code sets the output signal to the sum of the squares of all the input signals.

```
%assign y = LibBlockOutputSignal(0, "", "", 0)
%<y> = 0;

%assign rollVars = ["U"]
%foreach port = block.NumDataInputPorts - 1
  %roll sigIdx=RollRegions, lcv = RollThreshold, block, ...
    "Roller", rollVars
  %assign u = LibBlockInputSignal(port, "", lcv, sigIdx)
  %<y> += %<u> * %<u>;
  %endroll
%endforeach
```

Since the first parameter of LibBlockInputSignal is 0-indexed, you must index the foreach loop to start from 0 and end at NumDataInputPorts-1.

User Control Variable (ucv) Handling. This is an advanced mode and generally not needed by S-function authors.

If `ucv != ""`, `LibBlockInputSignal` returns an `rvalue` for the input signal using the user control variable indexing expression. The control variable indexing expression has the following form.

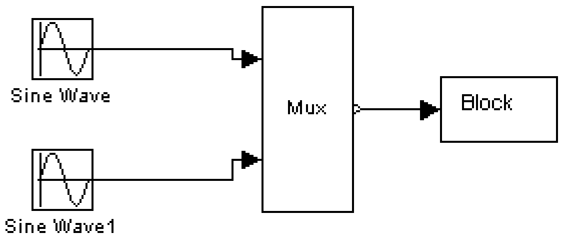
- `rvalue_id[%<ucv>]%<optional_real_or_imag_part>`

`rvalue_id` is obtained by looking at the integer part of `sigIdx`. Specifying `sigIdx` is required because the input to this block can be discontinuous, meaning that the input can come from several different memory areas (signal sources) and `sigIdx` is used to identify the area of interest for the `ucv`. Also, `sigIdx` is used to determine whether the real or imaginary part of a signal is to be accessed.

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

Note: the value for `lcv` is ignored and `sigIdx` must point to the same element in the input signal to which the `ucv` initially points.

The handling of `ucv` with `LibBlockInputSignal` requires care. Consider a discontinuous input signal feeding an input port as in the following block diagram.



To use `ucv` in a robust manner, you must use the `%roll` directive with a roll threshold of `1` and a Roller TLC file that has no loop header/trailer setup for this input signal. In addition, you need to use `ROLL_ITERATIONS` to determine the width of the current roll region, as in the following TLC code.

```
{
int i;
%assign rollVars  = [""]
%assign threshold = 1
  %roll sigIdx=RollRegions, lcv=threshold, block, ...
    "FlatRoller", rollVars
  %assign u = LibBlockInputSignal( 0, "i", "", sigIdx)
  %assign y = LibBlockOutputSignal(0, "i+%<sigIdx>", "", sigIdx)
  %assign p = LibBlockParameter( 0, "i+%<sigIdx>", "", sigIdx)
  for (i = 0; i < %<ROLL_ITERATIONS()>; i++) {
    %<y> = %<p> * %<u>;
  }
%endroll
}
```

Note, the `FlatRoller` has no loop header/trailer setup (`rollVars` is ignored). Its purpose is to walk the `RollRegions` of the block.

Alternatively, you can force a contiguous input signal to your block by specifying

```
ssSetInputPortRequiredContiguous(S, port, TRUE)
```

in your S-function.

In this case, the TLC code simplifies to

```
{
%assign u = LibBlockInputSignal( 0, "i", "", 0)
%assign y = LibBlockOutputSignal(0, "i", "", 0)
%assign p = LibBlockParameter(   0, "i", "", 0)

for (i = 0; i < %<DataInputPort[0].Width>; i++) {
  %<y> = %<p> * %<u>;
  }
}
```

If you create your own roller and the indexing does not conform to the way the Roller TLC file provided by the MathWorks operates, then you will need to use `ucv` instead of `lcv`.

Input Arguments (ucv, lcv, and sigIdx) Handling

Consider the following cases :

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

The value returned depends on what the input signal is connected to in the block diagram and how the function is invoked (e.g. in a %roll or directly). In the above example:

- Cases 1 and 2 occur when an explicit call is made with the `ucv` set to `"i"`.

  Case 1 occurs when `sigIdx` points to the block I/O vector, i.e., the first element that `"i"` starts with. For example, if you initialize `"i"` to be starting at offset `5`, then you should specify `sigIdx == 5`.

  Case 2 occurs when `sigIdx` pointing to the external input vector, i.e., the first element that `"i"` starts with. For example, if you initialize `"i"` to be starting at offset `20`, then you should specify `sigIdx == 20`.
- Cases 3 and 4 receive the same arguments, `lcv` and `sigIdx`, however, they produce different return values.

  Case 3 occurs when `LibBlockInputSignal` is called within a `%roll` directive and the current roll region is being rolled (`lcv != ""`).

  Case 4 occurs when LibBlockInputSignal is called within a `%roll` directive and the current roll region is not being rolled (`lcv == ""`).

When called within a `%roll` directive, this function looks at `ucv`, `lcv`, and `sigIdx`, the current roll region, and the current roll threshold to determine the return value. The variable `ucv` has highest precedence, `lcv` has the next highest precedence, and `sigIdx` has the lowest precedence. That is, if `ucv` is specified, it will be used (thus, when called in a `%roll` directive it is usually `""`). If `ucv` is not specified and `lcv` and `sigIdx` are specified, the returned value depends on whether or not the current roll region is being placed in a `for` loop or being expanded. If the roll region is being placed in a loop, then `lcv` is used, otherwise, `sigIdx` is used.

A direct call to this function (inside or outside of a `%roll` directive) will use `sigIdx` when `ucv` and `lcv` are specified as `""`.

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

LibBlockInputSignalAddr(portIdx, ucv, lcv, sigIdx)

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

When you need an input signal address, you must use this function instead of appending an “&” to the string returned by `LibBlockInputSignal`. For example, `LibBlockInputSignal` can return a literal constant, such as `5` (i.e., an invariant input signal). Real-Time Workshop tracks when `LibBlockInputSignalAddr` is called on an invariant signal and declares the signal as “const” data (which is addressable), instead of being placed as a literal constant in the generated code (which is not addressable).

Note, unlike `LibBlockInputSignal()`, the last input argument, `sigIdx`, is not overloaded. Hence, if the input signal is complex, the address of the complex container is returned.

Example

To get the address of a wide input signal and pass it to a user-function for processing, you could use

```
%assign uAddr = LibBlockInputSignalAddr(0, "", "", 0)
%assign y = LibBlockOutputSignal(0, "", "", 0)
%<y> = myfcn(%<uAddr>);
```

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalConnected(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalDataTypeId(portIdx)

Returns the numeric identifier (`id`) corresponding to the data type of the specified block input port.

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalDataTypeName(portIdx, reim)

Returns the name of the data type (e.g., `int_T`, ... `creal_T`) corresponding to the specified block input port.

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

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

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

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalDimensions(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalIsComplex(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalIsFrameData(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalLocalSampleTimeIndex(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalNumDimensions(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalOffsetTime(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalSampleTime(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalSampleTimeIndex(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalWidth(portIdx)

Returns the width of the specified block input port index.

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

The returned value is a valid `lvalue` (left-side value) for an expression. The block output destination can be a location in the block I/O vector (another block's input), the state vector, or an external output.

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

Real-Time Workshop tracks when a variable (e.g., a signal or parameter) is accessed by its address. To access the address of an output signal, use `LibBlockOutputSignalAddr` as in the following example.

```
%assign yAddr = LibBlockOutputSignalAddr(0, "", lcv, sigIdx)
x = %<yAddr>;
```

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalAddr(portIdx, ucv, lcv, sigIdx)

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

When an output signal address is needed, you must use this function instead of taking the address that is returned by `LibBlockOutputSignal`. For example, `LibBlockOutputSignal` can return a literal constant, such as 5 (i.e., an invariant output signal). When `LibBlockOutputSignalAddr` is called on an invariant signal, the signal is declared as a "const" instead of being placed as a literal constant in the generated code.

Note, unlike `LibBlockOutputSignal()`, the last argument, `sigIdx`, is not overloaded. Hence, if the output signal is complex, the address of the complex container is returned.

Example

To get the address of a wide output signal and pass it to a user-function for processing, you could use

```
%assign u = LibBlockOutputSignalAddr(0, "", "", 0)
%assign y = LibBlockOutputSignal(0, "", "", 0)
%<y> = myfcn (%<u>);
```

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

LibBlockOutputSignalBeingMerged(portIdx)

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

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

LibBlockOutputSignalConnected(portIdx)

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

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

LibBlockOutputSignalDataTypeId(portIdx)

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

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

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

LibBlockOutputSignalDataTypeName(portIdx, reim)

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

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

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

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

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalDimensions(portIdx)

Returns the dimensions of specified block output port.

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalIsComplex(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalIsFrameData(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalLocalSampleTimeIndex(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalNumDimensions(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalOffsetTime(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalSampleTime(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalSampleTimeIndex(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalWidth(portIdx)

Returns the width of specified block output port.

See function in *`matlabroot`*`/rtw/c/tlc/lib/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 `rlcv` or `clcv` is not equal to `""`).

The row and column index arguments are similar to the arguments for `LibBlockParameter`. The column index (`cidx`) is overloaded to handle complex numbers.

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

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

Returns the address of a matrix parameter.

Note `LibBlockMatrixParameterAddr` returns the address of a matrix parameter. Loop rolling is not supported (i.e., `rlcv` and `clcv` should both be the empty string).

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

LibBlockMatrixParameterBaseAddr(param)

Returns the base address of a matrix parameter.

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

LibBlockParameter(param, ucv, lcv, sigIdx)

Based on the parameter reference (param), the user control variable (ucv), the loop control variable (lcv), the signal index (sigIdx), and the state of parameter inlining, this function returns the appropriate reference to a block parameter.

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

Case	Function Call	May Produce
1	`LibBlockParameter(Gain, "i", lcv, sigIdx)`	`rtP.blockname[i]`
2	`LibBlockParameter(Gain, "i", lcv, sigIdx)`	`rtP.blockname`
3	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`p_Gain[i]`
4	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`p_Gain`
5	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`4.55`
6	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`rtP.blockname.re`
7	`LibBlockParameter(Gain, "", lcv, sigIdx)`	`rtP.blockname.im`

To illustrate the basic workings of this function, assume a noncomplex vector signal where Gain[0]=4.55:

```
LibBlockParameter(Gain, "", "i", 0)
```

Case	Rolling	Inline Parameter	Type	Result	Required In Memory
1	0	yes	scalar	`4.55`	no
2	1	yes	scalar	`4.55`	no
3	0	yes	vector	`4.55`	no
4	1	yes	vector	`p_Gain[i]`	yes
5	0	no	scalar	`rtP.blk.Gain`	no

Case	Rolling	Inline Parameter	Type	Result	Required In Memory
6	0	no	scalar	`rtP.blk.Gain`	no
7	0	no	vector	`rtP.blk.prm[0]`	no
8	0	no	vector	`p.Gain[i]`	yes

Note case 4. Even though inline parameter is true, the parameter must be placed in memory (RAM) since it's accessed inside a `for`-loop.

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

For example, if the parameter field had the MATLAB expression `'2*a'`, this function will return the C expression `'(2 * a)'`. The list of functions supported by this function is determined by the functions `FcnConvertNodeToExpr` and `FcnConvertIdToFcn`. To enhance functionality, augment or update either of these functions.

Note that certain types of expressions are not supported such as `x * y` where *both* `x` and `y` are nonscalars.

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

Warning

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

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

LibBlockParameterAddr(param, ucv, lcv, idx)

Returns the address of a block parameter.

Using `LibBlockParameterAddr` to access a parameter when the global `InlineParameters` variable is equal to 1 will cause the variable to be declared “const” in RAM instead of being inlined.

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

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

LibBlockParameterBaseAddr(param)

Returns the base address of a block parameter.

Using LibBlockParameterBaseAddr to access a parameter when the global `InlineParameters` variable is equal to one will cause the variable to be declared `"const"` in RAM instead of being inlined.

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

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

LibBlockParameterDataTypeId(param)

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

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

LibBlockParameterDataTypeName(param, reim)

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

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

LibBlockParameterDimensions(param)

Returns a row vector of length `N` (where `N >= 1`) giving the dimensions of the parameter data.

For example:

```
%assign dims  = LibBlockParameterDimensions("paramName")
%assign nDims = SIZE(dims,1)
%foreach i=nDims
   /* Dimension %<i+1> = %<dims[i]> */
%endforeach
```

This function differs from `LibBlockParameterSize` in that it returns the dimensions of the parameter data prior to collapsing the `Matrix` parameter to a column-major vector. The collapsing occurs for run-time parameters that have specified their `outputAsMatrix` field as `False`.

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

LibBlockParameterIsComplex(param)

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

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

LibBlockParameterSize(param)

Returns a vector of size 2 in the format `[nRows, nCols]` where `nRows` is the number of rows and `nCols` is the number of columns.

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

Block State and Work Vector Functions

LibBlockContinuousState(ucv, lcv, idx)

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

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

LibBlockDWork(dwork, ucv, lcv, sigIdx)

Returns a string corresponding to the specified block `DWORK` element.

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

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

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

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

If either `ucv` or `lcv` is specified (i.e., it is not equal to `""`) then the index part of the last input argument (`sigIdx`) is ignored.

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

LibBlockDWorkAddr(dwork, ucv, lcv, idx)

Returns a string corresponding to the address of the specified block `DWORK` element.

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

LibBlockDWorkDataTypeId(dwork)

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

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

LibBlockDWorkDataTypeName(dwork, reim)

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

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

LibBlockDWorkIsComplex(dwork)

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

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

LibBlockDWorkName(dwork)

Returns the name of the specified block `DWORK`.

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

LibBlockDWorkStorageClass(dwork)

Returns the storage class of specified block `DWORK`.

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

LibBlockDWorkUsedAsDiscreteState(dwork)

Returns 1 if the specified block `DWORK` is used as a discrete state, returns 0 otherwise.

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

LibBlockDWorkWidth(dwork)

Returns the width of the specified block `DWORK`.

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

LibBlockDiscreteState(ucv, lcv, idx)

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

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

LibBlockIWork(definediwork, ucv, lcv, idx)

Returns a string corresponding to the specified block `IWORK` element. See `LibBlockRWork()`

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

LibBlockMode(ucv, lcv, idx)

Returns a string corresponding to the specified block MODE element.

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

LibBlockNonSampledZC(ucv, lcv, NonSampledZCIdx)

Returns a string corresponding to the specified block NonSampledZC.

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

LibBlockPWork(definedpwork, ucv, lcv, idx)

Returns a string corresponding to the specified block PWORK element. See LibBlockRWork().

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

LibBlockRWork(definedrwork, ucv, lcv, idx)

Returns a string corresponding to the specified block RWORK element. The first argument, definedrwork, would typically be a symbol defined in the mdlRTW() routine of the C MEX file with something like the code below.

```
   ssWriteRTWWorkVect([...], "RWork", [...], "MyRWorkName", [...])
```

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

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

Block Path and Error Reporting Functions

LibBlockReportError(block,errorstring)

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

This function can be called with or without the block record scoped. To call this function without a block record scoped, pass the block record. To call this function when the block is scoped, pass `block = []`. Specifically

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

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

LibBlockReportFatalError(block,errorstring)

This should be used when reporting fatal (assert) errors for a block. Use this function for defensive programming. Refer to Appendix B, "TLC Error Handling."

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

LibBlockReportWarning(block,warnstring)

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

This function can be called with or without the block record scoped. To call this function without a block record scoped, pass the block record. To call this function when the block is scoped, pass `block = []`.

Specifically

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

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

LibGetBlockPath(block)

`LibGetBlockPath` returns the full block path name string for a block record, including carriage returns and other special characters that may be present in the name. Currently, the only other special string sequences defined are `'/*'` and `'*/'`.

The full block path name string is useful when accessing blocks from MATLAB. For example, you can use the full block name with `hilite_system()` via `FEVAL` to match the Simulink path name exactly.

Use `LibGetFormattedBlockPath` to get a block path suitable for placing in a comment or error message.

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

LibGetFormattedBlockPath(block)

`LibGetFormattedBlockPath` returns the full path name string of a block without any special characters. The string returned from this function is suitable for placing the block name, in comments or generated code, on a single line.

Currently, the special characters are carriage returns, `'/*'`, and `'*/'`. A carriage return is converted to a space, `'/*'` is converted to `'/+'`, and `'*/'` is converted to `'+/'`. Note that a `'/'` in the name is automatically converted to 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/lib/utillib.tlc`.

Code Configuration Functions

LibAddToCommonIncludes(incFileName)

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

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

Example:

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

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

LibAddToModelSources(newFile)

This function serves two purposes:

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

For inlined S-functions, `LibAddToModelSources` is generally called from `BlockTypeSetup`. This function adds a file name to the list of sources needed to build this model. This functions returns `1` if the filename passed in was a duplicate (i.e. it was already in the sources list) and `0` if it was not a duplicate.

As an S-function author, we recommend using the SFunctionModules block parameter instead of this function. See Writing S-functions.

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

LibCacheDefine(buffer)

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

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

Example.

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

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

LibCacheExtern(buffer)

LibCacheExtern should be called from inside BlockTypeSetup to cache an extern statement. Each call to this function appends your buffer to the existing cache buffer. The extern statements are placed in model.h.

Example

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

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

LibCacheFunctionPrototype(buffer)

LibCacheFunctionPrototype should be called from inside BlockTypeSetup to cache a function prototype. Each call to this function appends your buffer to the existing cache buffer. The prototypes are placed inside model.h.

Example

```
%openfile buffer
 extern int_T fun1(real_T x);
 extern real_T fun2(real_T y, int_T i);
%closefile buffer
```

```
%<LibCacheFunctionPrototype(buffer)>
```

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

LibCacheIncludes(buffer)

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

Example

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

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

LibCacheTypedefs(buffer)

LibCacheTypedefs should be called from inside BlockTypeSetup to cache typedef declarations. Each call to this function appends your buffer to the existing cache buffer. The typedef statements are placed inside model.h (or model_common.h).

Example. `%`

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

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

LibRegisterGNUMathFcnPrototypes()

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

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

LibRegisterISOCMathFcnPrototypes()

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

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

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

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

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

Sample Time Functions

LibBlockSampleTime(block)

Returns the block's sample time. The returned value depends on the sample time classification of the block, as shown in the following table.

Block Classification	Returned Value
Discrete	The actual sample time of a block (a real number greater than `0`.)
Continuous	`0.0`
Triggered	`-1.0`
Constant	`-2.0`

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

LibGetGlobalTIDFromLocalSFcnTID(sfcnTID)

Returns the model task identifier (sample time index) corresponding to the specified local S-function task identifier or port sample time. This function allows you to use one function to determine a global TID, independent of port- or block-based sample times.

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

The input argument to this function should be either

- `sfcnTID`: integer (e.g., 2)

  For block-based sample times (e.g., in S-function `mdlInitializeSizes`, `ssSetNumSampleTimes(S,N)` with `N > 1` was specified), `sfcnTID` is an integer starting at 0 of the corresponding local S-function sample time.

- or `sfcnTID`: string of the form `"InputPortIdxI", "OutputPortIdxI"` where `I` is a number ranging from 0 to the number of ports (e.g., `"InputPortIdx0"`, `"OutputPortIdx7"`). For port-based sample times (e.g., in S-function `mdlInitializeSizes`, `ssSetNumSampleTimes(S,PORT_BASED_SAMPLE_TIMES)` was specified), `sfcnTID` is a string giving the input (or output) port index.

Examples

Multirate block.

```
%assign globalTID = LibGetGlobalTIDFromLocalSFcnTID(2)
```

or

```
%assign globalTID =
LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx4")

%assign period =
CompiledModel.SampleTime[globalTID].PeriodAndOffset[0]
%assign offset =
CompiledModel.SampleTime[globalTID].PeriodAndOffset[1]
```

Inherited sample time block.

```
%switch (LibGetSFcnTIDType(0))
  %case "discrete"
  %case "continuous"
    %assign globalTID = LibGetGlobalTIDFromLocalSFcnTID(2)
    %assign period = ...
      CompiledModel.SampleTime[globalTID].PeriodAndOffset[0]
    %assign offset = ...
      CompiledModel.SampleTime[globalTID].PeriodAndOffset[1]
    %breaksw
  %case "triggered"
    %assign period = -1
    %assign offset = -1
    %breaksw
  %case "constant"
    %assign period = rtInf
    %assign offset = 0
    %breaksw
  %default
    %<LibBlockReportFatalError([],"Unknown tid type")>
%endswitch
```

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

LibGetNumSFcnSampleTimes(block)

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

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

LibGetSFcnTIDType(sfcnTID)

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

`"continuous"` if the specified `sfcnTID` is continuous.

`"discrete"` if the specified `sfcnTID` is discrete.

`"triggered"` if the specified `sfcnTID` is triggered.

`"constant"` if the specified `sfcnTID` is constant.

The format of `sfcnTID` must be the same as for `LibIsSFcnSampleHit`.

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

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

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/lib/utillib.tlc`.

LibIsContinuous(TID)

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

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

LibIsDiscrete(TID)

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

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

LibIsSFcnSampleHit(sfcnTID)

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

The input argument to this function should be either

- `sfcnTID`: integer (e.g., 2)

  For block-based sample times (e.g., in S-function `mdlInitializeSizes`, `ssSetNumSampleTimes(S,N)` with N > 1 was specified), `sfcnTID` is an integer starting at 0 of the corresponding local S-function sample time.

- or `sfcnTID`: `"InputPortIdx`*`I`*`", "OutputPortIdx`*`I`*`"` (e.g., `"InputPortIdx`*`0`*`"`, `"OutputPortIdx7"`)

  For port based sample times (e.g., in S-function `mdlInitializeSizes`, `ssSetNumSampleTimes(S,PORT_BASED_SAMPLE_TIMES)` was specified), `sfcnTID` is a string giving the input (or output) port index.

Examples

- Consider a multirate S-function block with 4 block sample times. The call `LibIsSFcnSampleHit(2)` will return the code to check for a sample hit on the third S-function block sample time.
- Consider a multirate S-function block with three input and eight output sample times. The call `LibIsSFcnSampleHit("InputPortIdx0")` returns the code to check for a sample hit on the first input port. The call `LibIsSFcnSampleHit("OutputPortIdx7")` returns the code to check for a sample hit on the eighth output port.

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

LibIsSFcnSpecialSampleHit(sfcnSTI, sfcnTID)

Returns the Simulink macro to promote a slow task (`sfcnSTI`) into a faster task (`sfcnTID`).

This advanced function is specifically intended for use in rate transition blocks. This function determines the global `TID` from the S-function `TID and` calls `LibIsSpecialSampleHit` using the global `TID`s for both the sample time index (`sti`) and the task ID (`tid`).

The input arguments to this function are

- For multirate S-function blocks:

  `sfcnSTI`: local S-function sample time index (`sti`) of the slow task that is to be promoted

  `sfcnTID`: local S-function task ID (`tid`) of the fast task where the slow task will be run.

- For single rate S-function blocks using `SS_OPTION_RATE_TRANSITION`, `sfcnSTI` and `sfcnTID` are ignored and should be specified as `""`.

The format of `sfcnSTI` and `sfcnTID` must follow that of the argument to `LibIsSFcnSampleHit`.

Examples

- A rate transition S-function (one sample time with `SS_OPTION_RATE_TRANSITION`)

```
if (%<LibIsSFcnSpecialSampleHit("","")>) {
```

- A multirate S-function with port-based sample times where the output rate is slower than the input rate (e.g., a zero-order hold operation)

```
if (%<LibIsSFcnSpecialSampleHit("OutputPortIdx0","InputPortIdx0")>) {
```

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

LibPortBasedSampleTimeBlockIsTriggered(block)

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

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

LibSetVarNextHitTime(block,tNext)

Generates code to set the next variable hit time. Blocks with variable sample time must call this function in their output functions.

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

Other Useful Functions

LibCallFCSS(system, simObject, portEl, tidVal)

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

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

See the `SFcnSystemOutputCall` record in the `model.rtw` file.

The return string is determined by the current code format.

Example

```
%foreach fcnCallIdx = NumSFcnSysOutputCalls
  %% call the downstream system
  %with SFcnSystemOutputCall[fcnCallIdx]
    %% skip unconnected function call outputs
    %if 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/fcncallgen.tlc`.

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

LibGenConstVectWithInit(data, typeId, varId)

Return an initialized static constant variable string of form:

```
static const typeName varId[] = { data };
```

The typeName is generated from typeId which can be one of:

```
tSS_DOUBLE, tSS_SINGLE, tSS_BOOLEAN, tSS_INT8, tSS_UINT8,
tSS_INT16, tSS_UINT16, tSS_INT32, tSS_UINT32,
```

The data input argument must be a numeric scalar or vector and must be finite (no Inf, -Inf, or NaN values).

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

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/lib/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/lib/dtypelib.tlc`.

LibGetDataTypeIdAliasedToFromId(id)

Return the data type `IdAliasedTo` corresponding to a data type ID.

See function in *`matlabroot`*`/rtw/c/tlc/lib/dtypelib.tlc`.

LibGetDataTypeIdResolvesToFromId(id)

Return the data type `IdResolvesTo` corresponding to a data type ID.

See function in *`matlabroot`*`/rtw/c/tlc/lib/dtypelib.tlc`.

LibGetDataTypeNameFromId(id)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/dtypelib.tlc`.

LibGetT()

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/lib/utillib.tlc`.

LibIsMajorTimeStep()

Returns a string to access whether the current simulation step is a major time step.

This function is the TLC version of the SimStruct macro: `ssIsMajorTimeStep`

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

LibIsMinorTimeStep()

Returns a string to access whether the current simulation step is a minor time step.

This function is the TLC version of the SimStruct macro `ssIsMinorTimeStep`

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

LibIsComplex(arg)

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

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

LibIsFirstInitCond(s)

`LibIsFirstInitCond` returns generated code intended for placement in the initialization function. This code determines, during run-time, whether the initialization function is being called for the first time.

This function also sets a flag that tells Real-Time Workshop if it needs to declare and maintain the `first-initialize-condition` flag.

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

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

LibMaxIntValue(dtype)

For a built-in integer data type, this function returns the formatted maximum value of that data type.

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

LibMinIntValue(dtype)

For a built-in integer data type, this function returns the formatted minimum value of that data type.

See function in *`matlabroot`*`/rtw/c/tlc/lib/dtypelib.tlc`.

Advanced Functions

LibBlockInputSignalBufferDstPort(portIdx)

Returns the output port corresponding to input port (`portIdx`) that share the same memory, otherwise (-1) is returned. You will need to use this function when you specify `ssSetInputPortOverWritable(S,portIdx,TRUE)` in your S-function.

If an input port and some output port of a block are

- Not test points, and
- The input port is overwritable

then the output port might reuse the same buffer as the input port. In this case, `LibBlockInputSignalBufferDstPort` returns the index of the output port that reuses the specified input port's buffer. If none of the block's output ports reuse the specified input port buffer, then this function returns `-1`.

This function is the TLC version of the Simulink macro `ssGetInputPortBufferDstPort`.

Example

Assume you have a block that has two input ports, both of which receive a complex number in 2-wide vectors. The block outputs the product of the two complex numbers.

```
%assign u1r = LibBlockInputSignal (0, "", "", 0)
%assign u1i = LibBlockInputSignal (0, "", "", 1)
%assign u2r = LibBlockInputSignal (1, "", "", 0)
%assign u2i = LibBlockInputSignal (1, "", "", 1)
%assign yr  = LibBlockOutputSignal (0, "", "", 0)
%assign yi  = LibBlockOutputSignal (0, "", "", 1)

%if (LibBlockInputSignalBufferDstPort(0) != -1)
  %% The first input is going to get overwritten by yr so
  %% we need to save the real part in a temporary variable.
  {
  real_T tmpRe = %<u1r>;
%assign u1r = "tmpRe";
%endif
%<yr> = %<u1r> * %<u2r> - %<u1i> * %<u2i>;
%<yi> = %<u1r> * %<u2i> + %<u1i> * %<u2r>;

%if (LibBlockInputSignalBufferDstPort(0) != -1)
  }
%endif
```

Note that this example could have equivalently used `(LibBlockInputSignalBufferDstPort(0) == 0)` as the Boolean condition for the `%if` statements since there is only one output port.

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalStorageClass(portIdx, idx)

Returns the storage class of the specified block input port signal. The storage class can be `"Auto"`, `"ExportedSignal"`, `"ImportedExtern"`, or `"ImportedExternPointer"`.

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockInputSignalStorageTypeQualifier(portIdx, idx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalIsGlobal(portIdx)

Returns 1 if the specified block output port signal is declared in the global scope, otherwise returns `0`.

If this function returns `1`, then the variable holding this signal is accessible from any where in generated code. For example, this function returns `1` for signals that are test points, external or invariant.

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalIsInBlockIO(portIdx)

Returns 1 if the specified block output port exists in the global Block I/O data structure. You may need to use this if you specify `ssSetOutputPortReusable(S,portIdx,TRUE)` in your S-function.

See *`matlabroot`*`/toolbox/simulink/blocks/tlc_c/sfun_multiport.tlc`.

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalIsValidLValue(portIdx)

Returns `1` if the specified block output port signal can be used as a valid left-side argument (`lvalue`) in an assignment expression, otherwise returns `0`. For example, this function returns `1` if the block output port signal is in read/write memory.

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalStorageClass(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockOutputSignalStorageTypeQualifier(portIdx)

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

See function in *`matlabroot`*`/rtw/c/tlc/lib/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
`"ExternalInput"`	If external input (root inport)
`"Ground"`	If unconnected or connected to ground

`"FcnCall"`	If function-call output
`0`	If not unique (i.e., source for a Merge block or a reused signal due to block I/O optimization)

Example

The following code fragment finds the block that drives the second input on the first port of the current block, then, assigns the input signal of this source block to the variable `y`:

```
%assign srcBlock = LibBlockSrcSignalBlock(0, 1)
%% Make sure that the source is a block
%if TYPE(srcBlock) == "Vector"
  %assign sys = srcBlock[0]
  %assign blk = srcBlock[1]
  %assign block = CompiledModel.System[sys].Block[blk]
  %with block
    %assign u = LibBlockInputSignal(0, "", "", 0)
    y = %<u>;
  %endwith
%endif
```

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

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/lib/blkiolib.tlc`.

LibBlockSrcSignalIsGlobalAndModifiable(portIdx, idx)

This function returns 1 if the source signal corresponding to the specified block input port element satisfies the following three conditions:

- It is readable everywhere in the generated code.

- It can be referenced by its address.
- Its value can change (i.e., it is not declared as a "`const`").

Otherwise, this function returns `0`.

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibBlockSrcSignalIsInvariant(portIdx, idx)

Returns `1` if the source signal corresponding to the specified block input port element is invariant (i.e., the signal does not change).

For example, a source block with a constant `TID` (or equivalently, an infinite sample time) would output an invariant signal.

See function in *`matlabroot`*`/rtw/c/tlc/lib/blkiolib.tlc`.

LibCreateHomogMathFcnRec(FcnName, FcnTypeId)

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

LibGetMathConstant(ConstName,ioTypeId)

Return a valid math constant expression with the proper datatype.

This function can only be called after `funclib.tlc` is included.

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

LibMathFcnExists(RTWFcnName, RTWFcnTypeId)

Return whether or not an implementation function exists for a given generic operation (function), given the specified function prototype.

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

LibSetMathFcnRecArgExpr(FcnRec, idx, argStr)

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

model.rtw

The following topics describe the structure and contents of *model*.rtw files, intermediate files which Real-Time Workshop compiles when generating code for a model, subsystem, or S-function.

model.rtw File Contents (p. A-3)	Description of *model*.rtw and its role in Real-Time Workshop
model.rtw Changes Between Real-Time Workshop 5.0 and 4.1 (p. A-12)	Highlights of changes to *model*.rtw since Release 4.1
General Information and Solver Specification (p. A-14)	How Simulink solver settings are reflected in *model*.rtw
RTWGenSettings Record (p. A-16)	How System Target File build settings are instantiated into *model*.rtw
Data Logging Information (DataLoggingOpts) (p. A-17)	How data logging information controlled by the **Workspace I/O** pane is stored in *model*.rtw
Data Structure Sizes (p. A-19)	Summary of the sizes of data structures written to *model*.rtw
Sample Time Information (p. A-21)	How *model*.rtw file describes model execution rates
Data Type Information (DataTypes record) (p. A-23)	Index of all possible data types that Simulink supports
Target Properties (p. A-25)	A reserved record that is subject to change and is used for tailoring targets
Block Type Counts (p. A-26)	Enumerates the number of each type of block in your model
External Inputs and Outputs (p. A-27)	Describes all root-level inport blocks and outport blocks
Block I/O Information (p. A-30)	Describes contents of the block I/O vector (rtB) structure
Data Type Work (DWork) Information (p. A-42)	Describes persistent memory storage for data types values other than REAL_T between consecutive time intervals

Continuous States (p. A-45)	Describes mapping between the continuous states and the continuous states structure (`rtX`)
Nonsampled Zero Crossings (NonsampledZCs) (p. A-46)	Describes nonsampled zero crossing structure and the nonsampled zero crossings direction structure
Zero Crossings Events (ZCEvents) (p. A-47)	Describes data for generating the `rtPZCE` (previous zero crossing event state) structure
Block Record Defaults (p. A-48)	Provides default values for block variables that are not explicitly written in the block records
Parameter Record Defaults (p. A-50)	Provides default values for block variables that are not explicitly written in the block parameter records
Data and Control Port Defaults (p. A-51)	Provides default values for block variables that are not explicitly written in the block data input port records.
Model Parameters Record (p. A-54)	Describes all block parameters found within the model
Reduced Blocks (p. A-58)	Identifies blocks which are non-operations during model execution
Custom Storage Class Record (p. A-59)	Identifies custom storage classes defined in the `Simulink.Parameter`, `Simulink.Signal`, and `Simulink.State` objects
Model Hierarchy (p. A-60)	Describe the model's hierarchy of blocks, signals, and subsystems
System Defaults (canonical inputs, outputs, and parameters) (p. A-63)	Describes the canonical input and output arguments of nonvirtual subsystems that are placed in functions
System Record (p. A-65)	Describes how to execute the blocks within your model, which can consist of multiple systems
Stateflow Record (p. A-88)	Contains one `Chart` record for each unique library chart used in the model
Model Checksums (p. A-89)	Provides hash-based checksums for the block parameter values, identifier names, and model structure
Block Specific Records (p. A-90)	Provides a block specific record for each Simulink built-in block and subsystem in the model
Object Records (p. A-118)	Provides mappings between `Simulink.Signals` and `Simulink.Parameters` objects

model.rtw File Contents

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.

This appendix is provided to help you modify existing block or target code generation, generate code for custom S-functions, or even create a new *code generator* to suit your needs. It describes the contents of the *model*.rtw file, which is created to represent your block diagram for the Real-Time Workshop build procedure, and is then 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 beyond signal, port, and block connectivity is excluded from the *model*.rtw file.

The general format of the *model*.rtw file is

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

Within CompiledModel, a series of hierarchical records map components of the model using *name-value* pair syntax. As in C programs, each record is delimited (scoped) by {...}, but unlike C code, *model*.rtw does not contain any functions or operators, only descriptors.

Using the model.rtw File

You need to understand the format and structure of the *model*.rtw file if you are writing a TLC script for an S-function (i.e., inlining the S-function), or if you wish to generate code in a language other than C. Most users only need to know a few of the details about the *model*.rtw file. For example, to inline 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.

Information such as signal connectivity and obtaining input and output connections for your S-function is encoded within the *model*.rtw file using mapping tables. Processing this information directly in the Target Language Compiler is difficult and may be handled differently from one release of our tools to another. To simplify writing TLC scripts for S-functions, and to provide compatibility between releases, many TLC library functions (which start with the prefix `Lib`) are provided. For example to access your inputs to your S-function, you should use `LibBlockInputSignal`. See Chapter 9, "TLC Function Library Reference," for a complete list of Target Language Compiler library functions.

When the Target Language Compiler calls the various functions that exist in your TLC script, 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 `mdlRTW` 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 `mdlRTW` for adding information to the *model*.rtw file. See *matlabroot*/simulink/src/sfuntmpl_doc.c for more information.

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

An example of a parameter value pair (also called a *field*) is

```
SigLabel "velocity"
```

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

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

Accessing Record Fields. To access fields within a record in a TLC script, use the *dot* (`.`) operator. For example, `Signal.SigLabel` 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 lets you use both relative and absolute scoping at will, similar to specifying files via absolute or relative pathnames. 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,

```
NumSignals 2
Signal {
  SigLabel "velocity"
}
Signal {
  SigLabel "position"
}
```

This code creates a parameter called `NumSignals` that specifies the length of the list. This is useful when using the `foreach` directive. To access the second signal, use `Signal[1]`. Note that TLC indexing is zero-based (the first index in a list is 0).

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

```
SignalDefaults {
  ComplexSignal no
}
Signal {
  SigLabel "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.

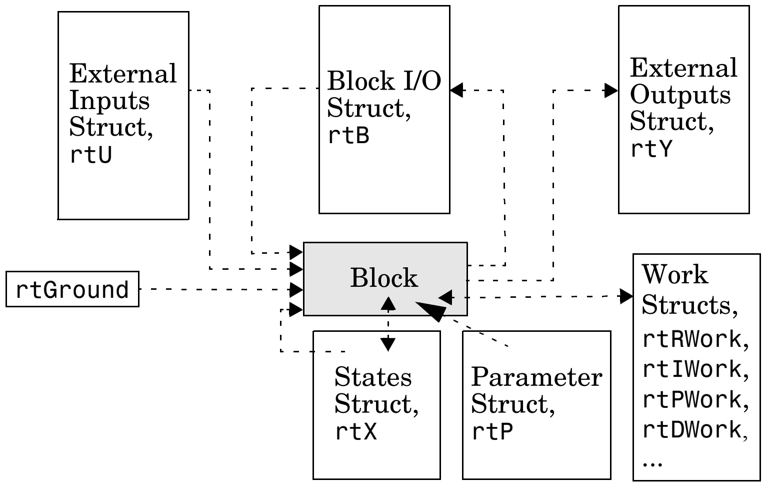
For example,

```
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 mirrors the structure of Simulink models. Conceptually, every model consists of systems and blocks. Blocks read their input, manage their states, and write their output. The figure below illustrates 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.


External Inputs Struct, rtU
Block I/O Struct, rtB
External Outputs Struct, rtY
rtGround
Block
Work Structs, rtRWork, rtIWork, rtPWork, rtDWork, ...
States Struct, rtX
Parameter Struct, rtP

Code Data Structure Concepts

In general, a block diagram is encoded using one or more of the following model data structures:

Table A-1: Model Data Structures

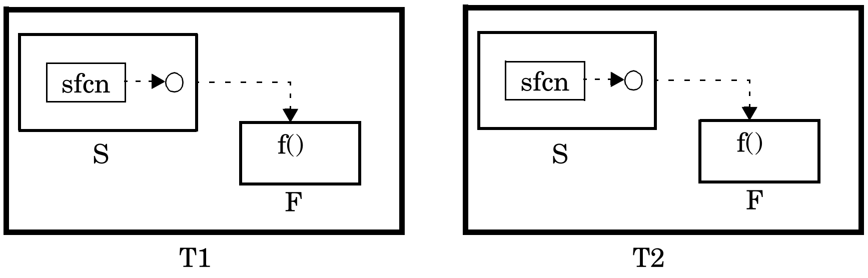
Data Structure	Contents
`rtB`	Block I/O. This is represented in the model.rtw file via the `CompiledModel.BlockOutputs` record.
`rtP`	Parameters structure (inline on and off). This is captured in the model.rtw file via the `CompileModel.ModelParameters` record.
`rtX`	Continuous states
`rtXdot`	Derivatives
`rtXdis`	Continuous state disabled setting
`rtZC`	Nonsampled zero crossings. Not related to `rtPZCE`.

Table A-1: Model Data Structures

rtZCdir	Nonsampled zero crossings direction. One-to-one mapping with rtZC. Note related to rtPZCE.
rtPZCE	Previous zero crossing event state. Used to detect edges in signals.

The model data structures are layered out according to the model hierarchy and the length is limited to at most 7 characters to allow us to stay within the 31 identifier character limit.

The layout includes function-call systems that reside in a parent system of their caller. For example, in the following model, subsystems T1 and T2 are structurally the same:



For this model, we have:

```
typedef struct rtB_S_tag {
  <block outputs in S>
} rtB_S;

typedef struct rtB_F_tag {
  <block outputs in F>
} rtB_F;

typedef struct rtB_T_tag {
  <regular (non system) block outputs in T>
  rtB_S  s;
  rtB_F  f;
} rtB_T;

typedef struct rtB {
  <regular (non system) block outputs in root>
  rtB_T t1;
  rtB_T t2;
}
```

This nesting of the block I/O data structure concept is applied to the other model structures (e.g., DWork). For the case of inline parameters off, the generated code will look roughly like:

```
F(f_input_signals, rtB_F *rtB, rtP_F *rtP)
{
  ...
}

S(s_input_signals      , rtB_S *rtB   , rtP_S *rtP,
  s_input_signals_for_F, rtB_F *rtB_F1, rtP_F *rtP_F1)
{
  ...
  F(f_input_signals, rtB_F1, rtP_F1)
  ...
}

T(t_input_signals, rtB_T *rtB, rtP_T *rtP)
{
  ...
  S(s_input_signals, &rtB->s, &rtP->s,
    s_input_signals_for_F, &rtB->f, &rtP->f)
  ...
}

MdlOutputs()
{
   ...
   T(t1_input_signals, &rtB.t1, &rtP.t1);  // outputs are passed via rtB.t1
   ...
   T(t2_input_signals, &rtB.t2, &rtP.t2);  // outputs are passed via rtB.t2
   ...
}
```

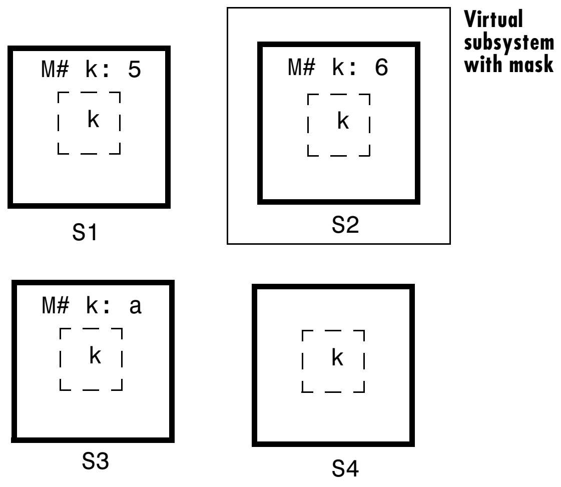
Inline Parameters On Code Structure

When inline parameters is on, Real-Time Workshop will generate the following structures:

- `rtP` — A semi-hierarchical structure for optimizing the placement of `auto` tunable variables in order to localize the tunable parameters within rtP (only systems that need to see a tunable variable are thus able to see it). The `auto` tunable variables consist of:
  - Workspace variables that are specified as tunable with storage class SimulinkGlobal (Auto) in the **Model Parameter Configuration** Dialog.
  - A Simulink.Parameter object is created with property `RTWInfo.StorageClass` set to `Auto`.

- `rtCP_`*`name`* — Individual variables for the #define/const parameters. These are run-time parameters that are inlined unless rolled. These are pooled together. Constant parameters not placed in a constant structure because this would disable parameter pooling and non-finite parameters cannot initialize at compile time.

Masks result in canonical input parameters. For example, consider:



```
>> a = 7      (tunable)
>> k = 8      (tunable)

S1: nonvirtual subsystem with a block accessing mask parameter k
    that has the value 5 in the mask field for k.
S2: nonvirtual subsystem within a virtual subsystem and S2
    contains a block accessing mask parameter k of the virtual
    subsystem that has the value 6 in the mask field for k.
S3: nonvirtual subsystem with a block accessing mask parameter k
    that has the tunable variable a in the mask field for k.
S4: nonvirtual subsystem that doesn't reside within a mask and
    is directly accessing tunable variable k.
```

The code will look like:

```
Sa(sa_input_signals, rtB_Sa *rtB, ..., param_type k)
   {
      // for S1, S2, S3
   }

Sb(sa_input_signals, rtB_Sa *rtB)
   {
     // for S4 (directly access rtP.k)
   }
```

model.rtw Changes Between Real-Time Workshop 5.0 and 4.1

This release consists of significant alterations to the *`model`*`.rtw` file.

The primary changes are:

- The number of `System` records has been reduced to `M`, where `M <= N`, and `N` is the number of actual systems in the model. This is the direct result of the code reuse capabilities introduced in Real-Time Workshop 5.0.
- The `Prototype` record in the `System` record was renamed to `Interface` to better reflect what it represents. In addition, many fields within the `Interface` record were added and others had name improvements.
- `DataTypes.SLName` was renamed to `DataTypes.DTName` to reflect that this is not a remapped Simulink name (`SLName`) due to special characters.
- The `DataStores` record was removed and this information has been incorporated into the `DWorks` record.
- Separate procedures, optionally in separate files, are always generated for systems that are not inlined. The generated code format for each system is based on the **RTW system code** setting in its associated subsystem block property dialog. Choices are:
  - `Auto` — Generate a reusable function if there is more than one reference to the system, and are structurally identical. If this is not possible, inline it.
  - `Inline` — Always inline the system (never reuse it).
  - `Reusable function` — Always generate a function for the system regardless of how many references exist to the system. This function includes arguments that are passed.
  - `Function` — Always generate a (non-reusable) function for the system regardless of how many references exist to the system. Such functions access the generated code global data structures, rtB, rtX, etc., and do not have arguments.

- - The following Real-Time Workshop data structures have been reworked to all become hierarchical based on the model system hierarchy.

Category	Data Structures Affected
`BlockOutputs`	`rtB`
`Parameters`	`rtP` (inline parameters `off` only)
`ContStates`	`rtX`, `rtXdot`, `rtXdis` (Note: `ssGetdX()`, `ssGetContStateDisabled()` were removed from direct reference)
`DWorks`	`rtDWork`
`NonsampledZC`	`rtZC`, `rtZCdir`
`PrevZCSigStates`	`rtPZCE` (previous zero crossing event state); was `rtPrevZCSigState`

- When Inline Parameters are on, we generate
  - `rtP` : A flat structure containing the ’auto’ tunable variables.
  - `rtCP_name` : Individual variables for the `#define`/`const` parameters (just like `rtC`).
- The rtC structure has been eliminated and replaced with individual references to the constant elements directly in the files corresponding to the systems that reference the invariant signals. Now, `#define` and `const` block I/O elements are handled in same manner.
- External inputs and outputs (`rtU`, `rtY`) are now created as flat structures.
- Removed `ChildSystemIndices` and `ChildSystemBlockIndices`. These have been replaced by `ChildSystems` within the scope of `System` records.

General Information and Solver Specification

Each *model*.rtw generated 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-2: Model.rtw General Information and Solver Specification

Variable/Record Name	Description
`Name`	Name of the Simulink model from which this *model*.rtw file was generated.
`Version`	Version of the *model*.rtw file.
`ModelVersion`	String that is equal to the Model Version property of the Model Info block.
`GeneratedOn`	Date and time when the *model*.rtw file was generated.
`ExprFolding`	0 or 1. Used in generating comments. If expression folding is active, comments incorporate multiple blocks.
`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.
`}`	

Table A-2: Model.rtw General Information and Solver Specification (Continued)

Variable/Record Name	Description
`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.
`Refine`	Refine factor.
`MaxStep`	Maximum step size.
`InitialStep`	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-3: Model.rtw RTWGenSettings Record

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

Note that the above RTWGenSettings records are examples only. All field names and values actually found in a *model*.rtw file are user-specified.

Data Logging Information (DataLoggingOpts)

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-4: Model.rtw Data Logging Information

Variable/Record Name	Description
`DataLoggingOpts {`	Data logging record describing the settings of Simulink simulation (Parameters, workspace, I/O settings).
`SaveFormat`	`0`, `1`, or `2` (or `Matrix`, `Structure`, or `StructureWithTime`, respectively)
`MaxRows`	Maximum number of rows or 0 for no limit.
`Decimation`	Data logging interval.
`TimeSaveName`	Name of time variable or `""` if not being logged.
`StateSaveName`	Name of state variable or `""` if not being logged.

Table A-4: 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`	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: `[sysIdx, blkIdx, blkStateIdx]` 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. The `blkStateIdx` will be: • -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 `blkStateIdxD` `Work` (data type work vector) of `blkIdx` block in `sysIdx` system Note that `sysIdx` and `blkIdx` are valid even if `blkStateIdx < 0`, and `N` is the number of signals being logged as states.
}	

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 are used, 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 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 sizes of data structures written to *`model`*`.rtw`.

Table A-5: Model.rtw Model Data Structure Sizes

Variable/Record Name	Description
`NumModelInputs`	Sum of all root-level import block widths. This is the length of the external input vector, `U`.
`NumModelOutputs`	Sum of all root-level outport block widths. This is the length of the external output vector, `Y`.
`NumNonVirtBlocksInModel`	Total number of nonvirtual blocks in the model.
`DirectFeedthrough`	Does the 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 (e.g., Integrator blocks) 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-5: Model.rtw Model Data Structure Sizes (Continued)

Variable/Record Name	Description
`NumNonsampledZCs`	Length of the model nonsampled zero-crossing vectors. There are two vectors of this length: the zero-crossing signals (`nonsampledZCs`), and the zero-crossing directions (`nonsampledZCdirs`). 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.
`NumDataStoreElements`	Total number of data store elements. This is the sum of the widths of all data store memory blocks in your model.
`NumBlockSignals`	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, `blockIO`.
`NumBlockParams`	Number of modifiable parameter elements (`params`). For example, the Gain block parameter contains modifiable parameter elements.
`NumAlgebraicLoops`	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 `FundamentalStepSize` 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-6: 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: • The number of system sample times is one (excluding triggered and 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.
`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.

Table A-6: Model.rtw Sample Times (Continued)

Variable/Record Name	Description
`SampleTime {`	One record for each sample time.
`  TID`	Task ID for this sample time.
`  PeriodAndOffset`	Period and offset for this sample time.
`}`	
`NumVariableSampleTimes`	The number of variable sample times in the model.

Data Type Information (DataTypes record)

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-7: Model.rtw Data Types

Variable/Record Name	Description
`DataTypes {`	Data types defining all built-in (`double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `fcn_call`, `integer`, `pointer`, `action`) 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.
`    DTName`	ASCII data type name.
`    Id`	Actual data type identifer which is used in Simulink. This is an integer that corresponds to the data type name.
`    IdAliasedTo`	Aliased-to data type identifier which is used in Simulink.
`    IdResolvesTo`	Resolves-to data type identifier which is used in Simulink.
`  }`	
`  IsSigned`	Integer (`0` or `1`) flag indicating that data type has sign bit
`  RequiredBits`	Minimum number of bits required to represent data type

Table A-7: Model.rtw Data Types

Variable/Record Name	Description
`FixedExp`	Fixed-point exponent (specifies binary point)
`FracSlope`	Fixed-point fractional slope
`Bias`	Fixed-point bias
`IsFixedPoint`	Integer (`0` or `1`) flag indicating that data type is fixed-point type
`}`	

Target Properties

The TargetProperties is a MathWorks reserved record that is subject to change and is used for tailoring targets.

Table A-8: Model.rtw TargetProperties

Variable/Record Name	Description
`TargetProperties {`	
`  HasObject`	0 or 1. Does this record contain a `Simulink.TargetProperites Object` record?
`  Object {`	
	Only present if `HasObject` is 1. This will be the contents of `TargetProperties` record. See "Object Records" on page A-118.
`  }`	
`}`	

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-9: Model.rtw Block Type Counts

Variable/Record Name	Description
`NumBlockTypeCounts`	Number of different types of nonvirtual blocks in your model. A block type correlates to the MATLAB command `get_param('`*`block`*`','BlockType')`.
`BlockTypeCount {`	One record for each block type count.
`  Type`	Type of the block (e.g., `Gain`).
`  Count`	Total number of the given type.
`}`	
`NumSFunctionNameCounts`	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.
`SFunctionNameCount {`	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.

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-10: Model.rtw External Inputs and Outputs

Variable/Record Name	Description
`ExternalInputs {`	
`ExternalInputDefaults {`	
`RecordType`	`ExternalInput`. The record type is used by TLC in general processing of the model data structures. This value is never overridden.
`Width`	1. The signal width of this external input.
`MemoryMapIdx`	`[-1,-1,-1]`: used to populate global data map record (GlobalMemoryMap) during code generation. Refer to *matlabroot*/rtw/c/tlc/mw/globalmaplib.tlc for usage information on the MemoryMapIdx field.
`HasObject`	0: Has a Simulink.Signal object that is used to map the external inputs to a specific location. This will be = 1 if this port is associated with a Simulink.Data object.
`DataTypeIdx`	0: The default signal data type is real_T.
`ComplexSignal`	no: The default signal is not complex.
`DirectFeedThrough`	yes: The default assumes the root inport requires its input.
`SigLabel`	"": Label on the output segment of the inport block.
`StorageClass`	Auto: The default value specifies that the Real-Time Workshop decides how external signals are declared.
`CustomStorageClassVersion`	0: Custom storage class version used

Table A-10: Model.rtw External Inputs and Outputs (Continued)

`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).
`BlockName`	Input port block name with special characters removed so it can be used in comments.
`Identifer`	Unique name across all external inputs to be used within the external input vector/structure.
`TID`	Integer task id (sample time index) giving the `SampleTime` record for this inport block.
`Width`	Signal width.
`Dimensions`	Vector of the form `[`*`nRows`*`,` *`nCols`*`]` for the signal. Only written if number of dimensions is greater than 1.
`DataTypeIdx`	Integer index of DataType record corresponding to this block. Only written if index is not 0.
`ComplexSignal`	`yes`: Only written if this inport signal is complex.
`SigLabel`	Signal label entered by user.
`DirectFeedThrough`	`no`: Only written if this inport doesn't require its signal when `MdlOutputs` is called.
`StorageClass`	Only written if not `Auto`. This setting determines how this signal is declared.
`StorageTypeQualifier`	Only written if not empty.
`HasObject`	`HasObject = 1` if this port is associated with a Simulink Data Object.
`Object {`	`Object` record is written if `HasObject = 1`. See “Object Information in the model.rtw File” in the *Real-Time Workshop User's Guide*. Also see “Object Records” on page A-118.
`}`	
`}`	

Table A-10: Model.rtw External Inputs and Outputs (Continued)

`ExternalOutputs {`	External outputs (root outports) from the block diagram.
`  ExternalOutputDefaults {`	
`    RecordType`	`ExternalOutput`. The record type is used by TLC in general processing of the model data structures. This value is never overridden.
`    Width`	`1`: Signal width (default is 1).
`    MemoryMapIdx`	`[-1,-1,-1]`: used to populate global data map record (`GlobalMemoryMap`) during code generation. Refer to *`matlabroot`*`/rtw/c/tlc/globalmaplib.tlc` for usage information on the `MemoryMapIdx` field.
`    SigLabel`	`""`: Label on the input segment of the outport block.
`    HasObject`	`0`: `HasObject = 1` if this port is associated with a Simulink Data Object.
`  }`	
`  NumExternalOutputs`	Number of `ExternalOutput` records that follow. This is equal to the number of root level outports.
`  ExternalOutput {`	One record per root-level outport block.
`    Block`	`[`*`sysIdx`*`, `*`blockIdx`*`]` of the outport block.
`    Width`	Signal width. Only present if not 1.
`    Sysidx`	`[externalOutputVectorIndex, SignalWidth]`
`    Dimensions`	Vector of the form `[`*`nRows`*`, `*`nCols`*`]` for the signal. Only written if number of dimensions is greater than 1.
`    SigLabel`	Label on the port signal, if any.
`  }`	
`}`	

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 {`	
`    RecordType`	`BlockOutput`.
`    SigSrc`	[]: The default source is non-existent indicating that signal reuse has caused this block output (memory location) to be written to by multiple sources.
`    Width`	1: Signal width

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

`MemoryMapIdx`	`[-1,-1,-1]`: used to populate global data map record (GlobalMemoryMap) during code generation. Refer to *`matlabroot`*`/rtw/c/tlc/mw/globalmaplib.tlc` for usage information on the `MemoryMapIdx` field.
`HasObject`	0: Indicates whether this signal has an associated `Simulink.Signal` object. Default is 0 indicating this signal output is not associated with a `Simulink.Signal`.
`TestPoint`	`no`: The default is that this signal has not been marked as a signal of interest in your model (see signal properties dialog).
`ConstExpr`	0: the default is this output is not a const expression.
`StorageClass`	`Auto`: The default value specifies that Real-Time Workshop decides where to declare this signal.
`StorageTypeQualifier`	`""`: The default type qualifier is empty.
`IdentiferScope`	`top-level`: The default is to declare the block output signal in the global block I/O vector. The other option is `fcn-level`, which will explicitly appear below.
`Invariant`	`no`: The default is that this signal has a non-constant sample time and change during execution.
`InitialValue`	`[]`: The default initial value is empty for non-invariant signals.
`DataTypeIdx`	`0`: The default data type is `real_T`.
`ComplexSignal`	`no`: The default is a non-complex real valued signal.
`SigLabel`	`""`: No signal label on the line.
`SigConnected`	`all`: All destination elements of the signal are connected to other nonvirtual blocks or root outports.
`NumReusedBlockOuputs`	0: Number of reused block outputs.
`NumMergedBlockOutputs`	0: Number of merged block outputs.
`}`	
`ReusedBlockOutputDefaults`	{

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

`RecordType`	`ReusedBlockOutput`
`SigLabel`	`""`: No signal label on the line.
`SigConnected`	`all`: All destination elements of signal are connected to other nonvirtual blocks or root outports.
`}`	
`MergedBlockOutputDefaults`	{
`RecordType`	`MergedBlockOutput`
`SigLabel`	`""`: No signal label on the line.
`SigConnected`	`all`: All destination elements of signal are connected to other nonvirtual blocks or root outports.
`}`	
`NumBlockOutputs`	Number of data output port signals.
`BlockOutput {`	One record for each data output signal. Remark: If a block outputs buffer contains the output of a Merge block then you will have `MergedBlockOutput` sub-sub-records inside a `ReusedBlockOutput` sub-record inside this `BlockOutput` record.
`Identifier`	Unique variable name across all block outputs.
`Width`	Output signal width, only written if its greater than 1.
`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`, `ImportedExternPointer or DefinedInTLC`. 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`.

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

`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 `InlineParameters=1`.
`InitialValue`	Non-empty vector that is only written when `Invariant` 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.
`SigSrc`	`[`*`systemIdx`*`, `*`callSiteIdx`*`, `*`blockIdx`*`, `*`outputPortIdx`*`]`. Note that *`callSiteIdx`* is an index into `System[`*`systemIdx`*`].CallSites`.
`SigLabel`	Signal label entered by user. Only written if non-empty (`""`).
`SigConnected`	Only written if one or more elements are not connected to destination non-virtual or root outport blocks. In this case it will be `none` if no elements are connected or a vector of length signal `Width` 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.
`MergedBlockOutput {`	Only written if the `BlockOutput` record corresponds to a Merge block. In this case, the number of `MergedBlockOutput` records is equal to the number of input ports on the Merge block.
`  Identifer`	Unique variable name across all block outputs.
`  SigSrc`	`[`*`systemIdx`*`, `*`callSiteIdx, blockIdx`*`, `*`outputPortIdx`*`]`. Note that *`callSiteIdx`* is an index into `System[`*`systemIdx`*`].CallSites`.
`  SigLabel`	Signal label entered by user. Only written if nonempty (`""`).
`}`	
`NumReusedBlockOutputs`	Number of `ReusedBlockOutput` records.

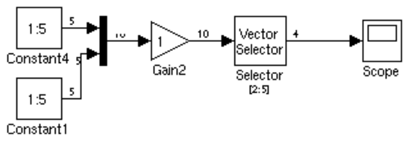
Table A-11: Model.rtw Block I/O Information (Continued)

`ReusedBlockOutput {`	Only written when this `BlockOutput` record is being reused by multiple blocks. There is one record for each block output port that is reused by this `BlockOutput` record.
`  Identifer`	Unique variable name across all block outputs.
`  SigSrc`	`[`*`systemIdx`*`, `*`callSiteIdx`*`, `*`blockIdx`*`, `*`outputPortIdx`*`]`. Note that *`callSiteIdx`* is an index into `System[`*`systemIdx`*`].CallSites`.
`  SigLabel`	Signal label entered by user. Only written if non-empty (`""`).
`  MergedBlockOutput {`	Only written if this `ReusedBlockOutput` record corresponds to a Merge block. In this case, the number of `MergedBlockOutput` records is equal to the number of input ports on the Merge block. See above for contents of the `MergedBlockOutput` records.
`  }`	
`}`	
`HasObject`	1: Indicates whether the output signal has an associated `Simulink.Signal` object. This only occurs when there is one source of this signal (i.e., not merged or reused).
`Object{`	Contents of `Simulink.Signal` Only written if `HasObject=1`, (a Unified Data Repository object is attached to the output signal) and the output signal is not being reused.
`  ...`	Fields in the `Object` record depend upon the contents of the object.
`}`	

```
    }
  }
}
```

Signal Connections

The example shows a model and its corresponding `BlockOuptuts` record and the way `SignalSrc` and `SignalOffset` with Block records is used to access signals.



The relevant `BlockOuptuts` and `Block` records in the model.rtw file and the way the signal feeding the Scope block is accessed are shown below.

```
BlockOuptuts {
...
BlockOutput {
      Identifier   Constant4
      Width        5
      TestPoint    yes
      SigSrc       [0, -1, 0, 0]
    }
    BlockOutput {
      Identifier   Constant1
      Width        5
      TestPoint    yes
      SigSrc       [0, -1, 1, 0]
    }
    BlockOutput {
      Identifier   Gain2
      Width        10
      TestPoint    yes
      SigSrc       [0, -1, 2, 0]
      SigConnected [0, 1@4, 0@5]
    }
}
...
```

```
Block {
   Type            Scope
   BlockIndex      [0, -1, 3]
   ...
   RollRegions      [0:3]
   NumDataInputPorts1
   DataInputPort {
       SignalSrc       [B2@4]
       SignalOffset    [1:4]
       Width       4
    }
}
```

To access the block I/O signal feeding the `Scope` block, we use its `DataInputPort` records `SignalSrc` and `SignalOffset`. `SignalSrc B2@4` refers to the `BlockOutput` index '2' and width of 4. **`BlockOutputs.BlockOutput[2]`**. Using `SignalOffset[i]`, i going from 0 to 3, we access the 2nd to 5th element in the 10 element vector from `Gain2`.(`rtB.Gain2[1]` to `rtB.Gain2[4]`).

Prior to TLC version 4.1, this was done using `SignalSrc[B11:B14]` and this will map throughout the `BlockOutputsMap` to the absolute offset in the `BlockOutput` record.

Using BlockOutput[i].SigSrc

SigSrc enables us to generate a reference to the block I/O "slot" for this record. The block I/O is a hierarchical structure definition.

Consider a model that has two references (`sysA1`,`sysA2`) to a nonvirtual subsystem named `sysA`. Within `sysA`, there are three references (`sysB1`,`sysB2`,`sysB3`) to `sysB`. Within `sysB`, there are five references (`sysC1`,`sysC2`,`sysC3`,`sysC4`,`sysC5`) to `sysC`. The total number of system instances:

```
sysA: 2
sysB: 2*3 = 6
sysC: 2*3*5 = 30
```

A model similar to this example is located at *`matlabroot`*`/toolbox/rtw/rtwdemos/tlctutorial/biohstruct/biohstruct_a2b3c5_ex.mdl`. Some examples based on this model:

- ex1: `rtB.sysA2.sysB1.sysC5.id` where sysA2 is the 2nd instance of a system with struct `rtB_sysA` in the root, `sysB1` is the 1st instance of `rtB_sysB`, in nonvirtual subsystem `sysA2`, and `sysC5` is the 5th instance of `rtB_sysC` in nonvirtual subsystem `sysB1`.
- ex2: `p->sysB1.sysC5.id` where pointer, `p` was previously set to `&rtB.sysA2`.
- ex3: `&p->sysB1.sysC5` to get at a structure address where, `p` was previously set to `&rtB.sysA2`.
- ex4: `&p->sysB1.sysC5.id[1]` to get at the 2nd element of an output vector in `sysC5`.

A simple TLC function that lets us generate these references is shown below. This function is located in
`matlabroot``/toolbox/rtw/rtwdemos/tlctutorial/biohstruct/biohstruct.tlc`.

```
%function GetBlockIoHStructAccess(blockIoIdx, stopAtSystemIdx, ...
  accessPrefix, accessBlockIoStructOnly)

  %assign bo          = CompiledModel.BlockOutputs.BlockOutput[blockIoIdx]
  %% bo.SigSrc = [systemIdx, callSiteIdx, blockIdx, outputPortIdx]
  %assign sysIdx      = bo.SigSrc[0]
  %assign callSiteIdx = bo.SigSrc[1]

  %assign ans         = accessBlockIoStructOnly ? "": bo.Identifier

  %foreach idx = CompiledModel.NumSystems
    %if sysIdx == stopAtSystemIdx
      %return accessPrefix + ans
    %else
      %% CallSites = Mx4 matrix where each row contains:
      %%    [callerSysIdx, callersCallSiteIdx, ...
      %%     graphicalSysIdx, ssBlkIdxInGraphicalSys]
      %assign callSiteRow = CompiledModel.System[sysIdx].CallSites[callSiteIdx]
      %%
      %% Locate the subsystem Block record to get at the call-site info
      %%
      %assign graphicalSysIdx        = callSiteRow[2]
      %assign ssBlkIdxInGraphicalSys = callSiteRow[3]
      %assign graphicalSys = CompiledModel.System[graphicalSysIdx]
      %assign ssBlk        = graphicalSys.Block[ssBlkIdxInGraphicalSys]
      %%
      %% Update the hierarchical strcture access.
      %%
      %assign ans = ssBlk.CallSiteInfo.StructId + "." + ans
      %%
      %% Walk up the call stack by moving sysIdx and callSiteIdx for
      %% next iteration
      %%
```

```
        %assign sysIdx       = callSiteRow[0]
        %assign callSiteIdx  = callSiteRow[1]
      %endif
    %endforeach

  %endfunction %% SLibGetBlockIOHStructAccess
```

This function lets us generate an observer reference to a particular slot in the block I/O (similar to ex1) by passing the desired `BlockOutput` record index, `stopAtSystemIdx` == the root system index, `accessPrefix == "rtB->"`, and `accessBlockIoStructOnly`==0.

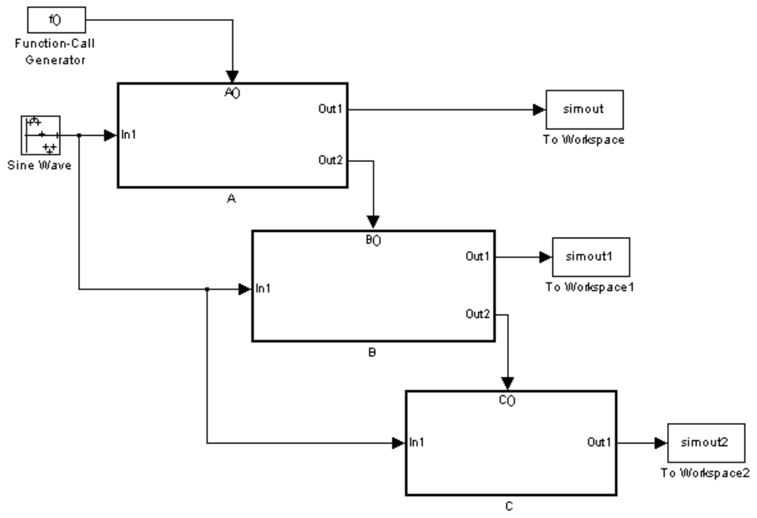
This function lets us generate internal references to the Block I/O structure (similar to ex2) where we are in a `System` record and need a portion of the fully qualified signal access. This is achieved by passing the desired `BlockOutput` record index, `stopAtSystemIdx == DeclSystemIdx` (a field within the system record), `accessPrefix == "p->"`, and `accessBlockIoStructOnly==0`.

This function lets us generate references similar to ex3 by calling by passing the first `BlockOutput` record index corresponding to the start of the desired structure, `stopAtSystemIdx == DeclSystemIdx`, `accessPrefix` = "&p->", and `accessBlockIoStructOnly==1`.

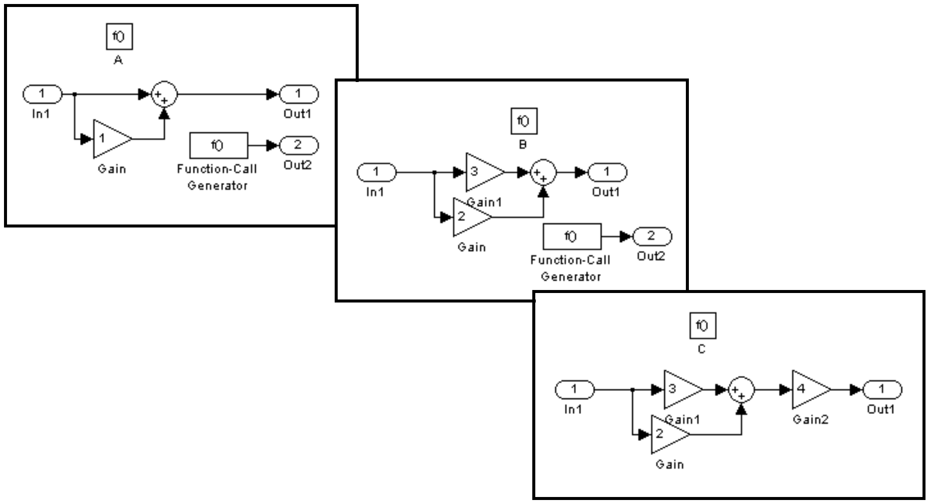
This function lets us generate references similar to ex4 by passing the BlockOutput record index corresponding `stopAtSystemIdx == DeclSystemIdx`, `accessPrefix == "&p->"`, and `accessBlockIoStructOnly==0`. The string "[1]" must then be appended to the return value of `GetBlockIoHStructAccess`. This example occurs when processing the canonical inputs to a system.

CallSites and their Interaction with SigSrc's.

We will now explore the CallSites matrix in a little more detail. Consider the model, which is located at:
`matlabroot``/toolbox/rtw/rtwdemos/tlctutorial/biohstruct/biohstruct_fcncall_ex.mdl`. The overall model is configured as shown below.



The model contains three function call subsystems, A, B, and C:



From Model.rtw System Record on page 65, the `System[i].CallSites` field contains

> *`[[callerSysIdx, callersCallSiteIdx, graphicalSysIdx,`*
> *`ssBlkIdxInGraphicalSys]; ...]`*

and from Model.rtw Block I/O Information on page 30 `BlockOuptuts.BlockOutput[i].SigSrc` field contains:

`[`*`systemIdx`*`, `*`callSiteIdx`*`, `*`blockIdx`*`, `*`outputPortIdx`*`]`

Looking at the `model.`*`rtw`* for this example, we see:

```
...
BlockOutputs {
...
  BlockOutput {                             << Accessed via BlockOutput[6]
      Identifier              Gain2
      SigSrc                  [0, 0, 3, 0]
  }
...
}
...
System {
    Type                    "function-call"
    Name                    "<Root>/C"
    ...
    CallSites               Matrix(1,4)
    [[1, 0, 3, 6];]
    ...
}
System {
    Type                    "function-call"
    Name                    "<Root>/B"
    ...
    CallSites               Matrix(1,4)
    [[2, 0, 3, 4];]
    ...
}
System {
    Type                    "function-call"
    Name                    "<Root>/A"
    ...
    CallSites               Matrix(1,4)
    [[3, -1, 3, 2];]
}
System {
    Type                    root
    Name                    "<Root>"
    ...
}
...
```

Consider a call to `GetBlockIoHStructAccess(6, 4, "rtB.", 0)` this will use the SigSrc record above to get at `sysIdx = 0` to access System with Name `"<Root>/C"` and `callSiteIdx = 0`. Then we enter the `%foreach` loop where we we access the `System[sysIdx].CallSites[callSiteIdx]` row. Notice that in System 'C' the *`callSysIdx`* (1 = System B) is not equal to the *`graphicalSysIdx`* (3 = System Root). This is because C is called by B, but resides in the Root System. This process continues and the function produces:

```
rtB.A.B.C.Gain2
```

Looking at the `CallSites` matrix, we see that the first two elements (*`callerSysIdx, callersCallSiteIdx`*) are used to construct the order in which the identifier path is created. The second two elements (*`graphicalSysIdx, ssBlkIdxInGraphicalSys`*) are used to obtain the specific name for each section of the identifier path.

Data Type Work (DWork) Information

Certain blocks require persistent memory 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.

Data Store Read and Write blocks access a Data Store Memory block. The memory associated with a Data Store Memory block is placed within a `DWork` record.

Note, all `RWork`, `IWork`, `PWork`, `ModeVector`, `DiscStates` 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
`DWorks {`	List of all data type work vectors in the model. There is one `DWork` record for each data type work vector in the model. The *source* of a data type work vector is a block. A block can have zero or more data type work vectors.
`  DWorkDefaults {`	Default values for the following `DWork` records.
`    RecordType`	`DWork`
`    Width`	`1`: Length of data type work vector.
`    MemoryMapIdx`	`[-1,-1,-1]`: used to populate global data map record (GlobalMemoryMap) during code generation. The TLC functions that are required to generate and access the global data map record are contained in *`matlabroot`*`/rtw/c/tlc/mw/globalmaplib.tlc`.
`    StorageClass`	`Auto`: Real-Time Workshop declares memory for this `DWork`.
`    CustomStorageClassVersion`	0

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

Variable/Record Name	Description
`StorageTypeQualifier`	`""`: No type qualifier.
`HasObject`	`0`: if `HasObject = 1`, this `DWork` is a Simulink Data Object. See "Object Information in the model.rtw File" in the *Real-Time Workshop User's Guide*.
`DataTypeIdx`	`0`: Default vector of a `DWork` record is a `real_T` vector.
`ComplexSignal`	`no`: Default vector of a `DWork` record is a non-complex vector.
`Origin`	`DWORK`: This datatype work vector originated as a `DWork`.
`UsedAs`	`DWORK`: This data type work vector is used as a `DWork`.
`InitialValue`	`[]`
`AcceleratorPadding`	0
`}`	
`NumDWorks`	Number of data type work vectors in the model. Each block can register 0 or more data type work vectors. This include discrete states, `RWork`, `IWork`, `PWork` and `ModeVector`. 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, `NumDWorks` is 2.
`DWork {`	One `DWork` record for each data type work vector.
`Identifier`	Identifier provides a unique variable name across all data type work vectors, can also be used to reference Simulink State objects.
`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).

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

Variable/Record Name	Description
`    ComplexSignal`	`yes`: Only written if this data type work vector is complex.
`    Origin`	Only written if not default (`DWORK`); it can be one of `MODE`, `RWORK`, `IWORK`, `PWORK` or `DSTATE`.
`    UsedAs`	Only written if not default (`DWORK`); it can be either `DSTATE` or `SCRATCH`.
`    SigSrc`	`[`*`systemIdx, callSiteIdx, blockIdex`*`,` *`dworkIdx`*`]`.
`    StorageClass`	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 *`model`*`.rtw` file.
`    StorageTypeQualifier`	Only written if non-empty (e.g., `const` or something similar).
`    HasObject`	`HasObject = 1` if this `DWork` is associated with a Simulink Data Object. In this case, an `Object` record is written. If `HasObject = 0`, no `Object` record will be written.
`    HasObject`	1: Indicates whether the output signal has an associated `Simulink.State` object. This only occurs when there is one source of this signal (i.e., not merged or reused).
`    Object{`	Contents of `Simulink.State`; only written if `HasObject=1`, (a Unified Data Repository object is attached to the output signal) and the output signal is not being reused.
`       ...`	Fields in the `Object` record depend upon the contents of the object.
`    }`	
`  }`	
`}`	

Continuous States

The `CompiledModel.ContStates` record is used to generate the continuous states structure (`rtX`) in the generated code. It contains a mapping between the blocks continuous state (`X`) and the continuous states structure (`rtX`).

Table A-13: Model.rtw State Mapping Information

Variable/Record Name	Description
`ContStates {`	
`  ContStateDefaults {`	
`    RecordType`	`ContState`
`    Width`	`1`
`    InitialValue`	`[0.0]`
`    StorageClass`	`Auto`
`}`	
`NumContStates`	ContState record list length (i.e., the number of blocks with continuous states).
`ContState {`	One record for each block with continuous states.
`  Identifier`	Unique name (identifier) within the current System record scope. This is always the terminal field for a state value.
`  Width`	Number of states the block has, if not default
`  SigSrc`	Block generating this state `[`*`systemIdx`*`, `*`callSiteIdx`*`, `*`blockIdx`*`]`
`  InitialValue`	A vector continaing the initial state values; present only when one or more initial state values are nonzero.
`}`	

Nonsampled Zero Crossings (NonsampledZCs)

The `CompiledModel.NonsampledZCs` record is used to generate the nonsampled zero crossing structure and the nonsampled zero crossings direction structure (`rtZC`, `rtZCdir`) in the generated code. It contains a mapping between the blocks nonsampled zero crossings and the model-wide nonsampled zero crossings structures. The number of nonsampled zero crossings will always be zero for fixed-step models.

Table A-14: Model.rtw State Mapping Information

Variable/Record Name	Description
`NonsampledZCs {`	
`  NonsampledZCDefaults {`	
`    Width`	`1`
`}`	
`NumNonsampledZCs`	NonsampledZC record list length (i.e., the number of blocks with nonsampled zero crossings).
`NonsampledZC {`	One for record for each block with nonsampled zero crossings.
`  Identifier`	Unique name (identifier) within the current System record scope. This is always the terminal field for a state value.
`  Width`	Number of nonsampled zero crossings the block has.
`  SigSrc`	`[`*`systemIdx`*`, `*`callSiteIdx`*`, `*`blockIdx`*`]` Block generating this nonsampled zero crossing
`  Direction`	A vector indicating the direction in which to search for zero crossings, containing values of `-1` (positive-to-negative), `0` (either direction), and `1` (negative-to-positive).
`}`	

Zero Crossings Events (ZCEvents)

The `CompiledModel.ZCEvents` is used to generate the `rtPZCE` (previous zero crossing event state) structure in the generated code. The data type of each field in this structure is `ZCSigState`, which encodes the last value of the signal we were tracking for a zero crossing in an enumeration (see *matlabroot*`/simulink/include/simstruc_types.h`).

Table A-15: Model.rtw State Mapping Information

Variable/Record Name	Description
`ZCEvents {`	
`  ZCEventDefaults {`	
`    Width`	`1`
`}`	
`NumZCEvents`	`ZCEvent` record list length (i.e., the number of blocks with zero crossing events).
`ZCEvent {`	One for record for each block with zero crossing events.
`  Identifier`	Unique name (identifier) within the current System record scope. This is always the terminal field for a state value.
`  Width`	Number of zero crossing events the block has.
`  SigSrc`	`[`*`systemIdx`*`, `*`callSiteIdx`*`, `*`blockIdx`*`]` Block generating this zero crossing event.
`}`	

Block Record Defaults

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

Table A-16: Model.rtw Block Defaults

Variable/Record Name	Description
BlockDefaults {	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.
RecordType	Block
InMask	no
AlgebraicLoopId	0
PortBasedSampleTimes	no
ContStates	[0,0,0]
ModeVector	[0,-1]
RWork	[0,-1]
IWork	[0,-1]
PWork	[0,-1]
DiscStates	[0,-1]
NumDWork	0
NumNonsampledZCs	0
ZCEvents	[0,0]
ModelEventSystemsToCall	[]
RollRegions	[]
NumDataInputPorts	0
NumControlInputPorts	0
NumDataOutputPorts	0
Parameters	[0,0]
NumParameterGroups	0
Description	[]
ReducedBlocksConnected	[]
TLCExprCompliant	0

Table A-16: Model.rtw Block Defaults

Variable/Record Name	Description
`CustomStorageClassCompliant`	1
`EnforceIntegerDowncast`	1
`HaveBlockComment`	0
`InFixptMode`	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-17: Model.rtw Parameter Defaults

Variable/Record Name	Description
`ParameterDefaults {`	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.
`MemoryMapIdx`	`[-1,-1,-1]`: used to populate global data map record (GlobalMemoryMap) during code generation. Refer to *`matlabroot`*`/rtw/c/tlc/mw/globalmaplib.tlc` for usage information on the `MemoryMapIdx` field.
`RecordType`	`BlockParameter`
`OriginalDataTypeIdx`	`0`: corresponds to real_T
`DataTypeIdx`	`0`: corresponds to `real_T`
`ComplexSignal`	`no`
`Width`	`1`
`Dimensions`	`[1, 1]`
`Tunable`	`off`: If inline parameters check box is off, otherwise `on` if inline parameters check box is on.
`StorageClass`	`Auto`
`NeedParenthesis`	`1`
`StringTransformed`	`""`: Transformed dialog parameter to be used as a string in the comment generated by the parameter
`Value`	`[0.0]`
}	

Data and Control Port Defaults

In the event that `DataInputPort`, `HiddenDataInputPort`, `ControlInputPort`, `DataOutputPort`, or `HiddenDataOutputPort` values are not provided in a block data record, the default values are used as provided by the following 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-18: Model.rtw Data and Control Input Port Defaults

Variable/Record Name	Description
`DataInputPortDefaults {` or `HiddenDataInputPortDefaults {`	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. Note, this record is duplicated and written as `HiddenDataInputPortDefaults` for the nonvirtual inport blocks which have `HiddenDataInputPort` records corresponding to their source.
`RecordType`	`DataInputPort`
`SignalSrc`	`[]`: `See “Signal Connections” on page A-35 for information on use of this field.`
`SignalOffset`	`[0]`: See “Signal Connections” on page A-35 for information on use of this field.
`Width`	`1`
`DataTypeIdx`	`0`
`SystemToCall`	`[-1, -1]` ([*`systemIdx, callSiteIdx`*] where -1 means no system to call.)
`ComplexSignal`	`no`
`FrameData`	`no`
`RecurseOnInput`	`0` (used for expression folding)

Table A-18: Model.rtw Data and Control Input Port Defaults (Continued)

Variable/Record Name	Description
`HaveGround`	`no`
`DirectFeedThrough`	`yes`. Only written if the `rtwgen` option `WriteBlockConnections` has been specified as on.
`BufferDstPort`	`-1`: Default is no output ports are reusing the corresponding input port buffer.
`AllowScalarExpandedExpr`	`0` (used for expression folding).
`}`	
`ControlInputPortDefaults {` or `Hidden ControlInputPortDefaults {`	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.
`RecordType`	`ControlInputPort`
`SignalSrc`	`[]: See "Signal Connections" on page A-35` for information on use of this field.
`SignalOffset`	`[0]`: See "Signal Connections" on page A-35 for information on use of this field.
`Width`	`1`
`DataTypeIdx`	`0`
`ComplexSignal`	`no`
`DirectFeedThrough`	`yes.` Only written if the rtwgen option `WriteBlockConnections` has been specified as on.
`FrameData`	`no`
`RecurseOnInput`	`0`
`HaveGround`	`no`

Table A-18: Model.rtw Data and Control Input Port Defaults (Continued)

Variable/Record Name	Description
`  DirectFeedThrough`	`yes`
`  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.
`  RecordType`	DataOutputPort
`  SignalSrc`	`[]`
`  SignalOffset`	`[0]`
`  Width`	`1`
`  DataTypeIdx`	`0`
`  ComplexSignal`	`no`
`  OutputExpression`	`0`
`  TrivialOutputExpression`	`0`
`  FrameData`	`no`
`  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-19: 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.
`  NumConstPrms`	Number of inlined parameter values. These are inlined, unless the roll threshold causes them to be placed in global constant memory. These parameters are possibly pooled across 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.
`  NumCustomStorageClass`	Number of parameters with custom (user specified) storage class. These parameters are derived from any `Simulink.Parameter`'s with properties `RTWInfo.StorageClass` set to 'Custom' and the corresponding `RTWInfo.CustomStorageClass` property giving the definition of the storage class. These parameters may be shared by multiple blocks.
`  ParameterDefaults {`	Default values for the following `Parameter` records.
`    RecordType`	ModelParameter
`    MemoryMapIdx`	[-1, -1, -1]. Undefined `GlobalMemoryMap` entry.

Table A-19: Model.rtw Model Parameters Record (Continued)

Variable/Record Name	Description
`HasObject`	0: Default is no `Simulink.Parameter` object is associated with this parameter.
`DataTypeIdx`	0: Default is `real_T` data type.
`OriginalDataTypeIdx`	0: Default is `real_T` data type.
`ComplexSignal`	no: Default is non-complex
`Width`	1
`Dimensions`	[1 1]
`Tunable`	`no`: Default value is not tunable.
`RollVarDeclared`	0: Default is Roll variable not declared for variable
`Transformed`	`no`: Default is that this parameter is not transformed and it maps back directly to dialog parameters.
`StorageClass`	`Auto`: Default value is `Auto` (Real-Time Workshop declares the memory).
`CustomStorageClassVersion`	0: Default version number of custom storage class
`TypeQualifier`	`""`: Default is no type qualifier.
`IsSfcnSizePrm`	0: Default is not an S-function sizes parameter (only used by non-inlined S-functions)
`SystemScoping`	[0, 0]: Default value for system scoping [`numOpens, numCloses`].
`Value`	0.0: Default value of parameter
`AcceleratorPadding`	0: Default is no padding added by Accelerator after parameter
`}`	
`Parameter {`	

Table A-19: Model.rtw Model Parameters Record (Continued)

Variable/Record Name	Description
`Identifier`	Identifier used in the generated code.
`LogicalSrc`	Parameter index
`Tunable`	If inlined parameters check box is off, then all parameter values are tunable (they will reside in the `rtP` 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).
`Transformed`	`yes`: If inline parameters is on and this is a run-time parameter that is obtained by applying a transformation to one or more dialog parameters.
`StorageClass`	Specifies where to declare/place this parameter value in memory (`Auto`, `Auto_SFCN`, `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`). Default value is `Auto` in which case this field is not written to the *`model`*`.rtw` file. `AUTO_SFCN` indicates that a tunable parameter is being passed to the S-function via its `ssGetSFcnParam`. This is for use by the S-function target only.
`TypeQualifier`	String used as a type qualifier for the declaration of the parameter (e.g., `static`).
`Value`	Evaluated value of this parameter.
`Width`	Total number of elements in parameter value
`Dimensions`	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.
`OriginalDataTypeIdx`	The original data type of this parameter. Real-Time Workshop supports a wide range of data types, including custom defined data types. For example, the original data type of a block run-time parameter from the Fixed-Point Blockset may be `sfix`*`name`* (or `ufix`*`name`*) and this parameter resolves to `int8_T`. In this case, the original data type is `sfix`*`name`* (or `ufix`*`name`*) and the data type is `int8_T`.

Table A-19: Model.rtw Model Parameters Record (Continued)

Variable/Record Name	Description
`DataTypeIdx`	Data type index into the data type table (`CompiledModel.DataTypes.DataType`).
`ComplexSignal`	`yes` or `no`, is this a complex signal?
`IsSfcnSizePrm`	1, if this is an S-function sizes parameter, 0 otherwise.
`SystemScoping`	A 2-D matrix that captures how deep the parameter is within the hierarchical data structure
`ReferencedBy`	A Nx4 matrix. Each row consists of *`[systemIdx, callSiteIdx, blockIdx, paramIdx]`* that identifies a usage of this parameter value. If N>1, then this parameter value is shared by multiple blocks.
`OwnerSysIdx`	A 2-D matrix that captures the system index and callsite index of the system in which the parameter is used.
`HasObject`	1, if this run-time parameter is from a `Simulink.Parametr`.
`Object {`	The `Simulink.Parameter`. Only present if `HasObject` == 1.
`}`	
`AcceleratorPadding`	Padding added after this parameter when using the Accelerator
`}`	
`}`	

Reduced Blocks

Nonvirtual blocks can specify that they are to be treated as a wire (i.e., a non-operation during model execution where the input is passed to the output). S-functions achieve this capability via the `ssSetBlockReduction(S,true)` macro.

Table A-20: Model.rtw Model Parameters Record

Variable/Record Name	Description
`ReducedBlocks {`	List of blocks that are being treated as a wire.
`NumReducedBlocks`	Integer, total number of blocks that are being treated as a wire.
`ReducedBlock {`	One record for each block that is being treated as a wire.
`Type`	Block type, e.g., S-function.
`Name`	Block name preceded with a `<root>` or `<S#>` token.
`OptimizationInfo`	Information about how/why this block was reduced.
`InComment`	0: Used for generating comments involving the blocks being treated as a wire. This is 0 initially, then during code generation, this will become 1.
`}`	

Custom Storage Class Record

Custom storage classes defined in the `Simulink.Parameter`, `Simulink.Signal`, and `Simulink.State` objects, enable the mapping of data in the generated code to user defined data structures.

Table A-21: Model.rtw Model Parameters Record

Variable/Record Name	Description
`CustomStorageClasses {`	List of blocks that are being treated as a wire.
`  NumCustomStorageClasses`	Length of the CustomStorageClass record list.
`  CustomStorageClass {`	One record for each custom storage class.
`  Name`	Storage class name
`  Package`	Package that the storage class is defined in.
`}`	

Model Hierarchy

The *root* is the top level of the block diagram. The `RootSignals` 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 `RootSignals` and `Subsystem` records enable you to reconstruct the graphical model hierarchy. This is useful for third party monitoring and parameter tuning tools.

Table A-22: Model.rtw Model Hierarchy (Blocks, Signals, and Subsystems)

Variable/Record Name	Description
`SignalDefaults {`	Default fields for `Signal`
`SigLabel`	`""`: Signal label
`OutputPort`	[0,1]: [`outputPortIndex`, `outputPortWidth`].
`SignalOffset`	[0]: Vector whose length is the width of the output port. Needs to be used with the corresponding element in `SignalSrc` to identify the correct signal location within the record identified by `SignalSrc`.
`}`	
`RootSignals {`	Signal and block information in the root window. The contents of this record is identical to `Subsystem` record, except the `SysId`, `Name`, `SLName`, and `Virtual` fields are not present.
`}`	
`NumSubsystems`	Total number of (non-empty) subsystems in the model. To obtain the number of virtual subsystems, use `CompiledModel.NumSubsystems - CompiledModel.NumSystems - 1`.
`Subsystem {`	One record for each subsystem.

Table A-22: Model.rtw Model Hierarchy (Blocks, Signals, and Subsystems) (Continued)

`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 *not* equal to `Name`. This will occur when generating code using the `rtwgen` `StringMapings` 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
	`my block name`
	`/* comment */`
	
	The *`model`*`.rtw` file will contain
	`Name   <Root>/my block name //+ comment +//`
	`SLName <Root>/my block name\n//* comment *//`

`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`	`[`*`systemIdx, callsiteIdx, blockIdx`*`]` or block name string if a virtual block.
`OutportName`	This field is written only if the signal is emanating from a subsystem. It is the Outport block name corresponding to the output signal of a subsystem block.
`SigLabel`	Signal label if present.
`OutputPort`	`[outputPortIndex, outputPortWidth]` of the block producing this signal.
`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`).

Table A-22: Model.rtw Model Hierarchy (Blocks, Signals, and Subsystems) (Continued)

`ComplexSignal`	`yes`: Only written for nonvirtual blocks and if signal is complex.
`SignalSrc`	Vector of length *`outputPortWith`* giving the location of the signal source. Used with `SignalOffset`, this will give an index into the block I/O vector (B), or state vector (X), or external input vector (U) or unconnected ground (G) or F indicating the source is a function call.
`SignalOffset`	Vector of length same as `SignalSrc`. Used with corresponding element in the `SignalSrc` field to identify the correct data offset in the record identified by `SignalSrc`.
`}`	
`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.
`}`	

System Defaults (canonical inputs, outputs, and parameters)

Default record field values for the canonical input and output arguments of nonvirtual subsystems that are placed in functions.

Table A-23: Model.rtw Data and Control Input Port Defaults

Variable/Record Name	Description
`CanonicalInputArgDefaults {`	
`  SignalSrc`	`[]`
`  SignalOffset`	`[0]`
`}`	
`CanonicalInputArgDefDefaults {`	
`  Identifier`	`""`
`  FirstSignalSrc`	`[]`
`  FirstSignalOffset`	`[0]`
`  SignalWidth`	`1`
`}`	
`CanonicalOutputArgDefaults {`	
`  SignalSrc`	`[]`
`  SignalOffset`	`[0]`
`}`	
`CanonicalOutputArgDefDefaults {`	
`  Identifier`	`""`

Table A-23: Model.rtw Data and Control Input Port Defaults (Continued)

Variable/Record Name	Description
`  FirstSignalSrc`	`[]`
`  FirstSignalOffset`	`[0]`
`  SignalWidth`	`1`
`}`	
`CanonicalPrmArgDefDefaults {`	
`  Dimensions`	`[1, 1]`
`  DataTypeIdx`	`0`
`  ComplexSignal`	`no`
`}`	

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.

There is one System record each system that is generated as a procedure (Inline==0). There is one System record for each instance of an inlined system (Inline==1). For example, if a nonvirtual subsystem block in a library is inlined an referenced five times, then there will be five System records for it.

During code generation, each system contains all relevant information to generate either the procedure bodies (or inlined code), and any required input argument types (e.g. the *rtB_name* typedef's structures) for the System.

Note, the various code generation data structures (e.g. *rtB_name*) are not flattened when a system is inlined, thus the overall structures (e.g. rtB) have the same general shape regardless of whether or not the system is inlined.

Table A-24: Model.rtw System Record

Variable/Record Name	Description
NumSystems	Total number of System records (non-virtual) in the model. Use CompiledModel.NumSystems -1 to get the number of nonvirtual subsystems.
System {	One for each system in the model. This is equal to NumNonvirtSubsystems plus 1 for the root system.
Type	root, atomic, enable, trigger, enable_with_trigger, function-call, action or iterator
CalledByBlock	yes if this is a derived atomic subsystem that is conditionally executed by another block, no otherwise. Think of this field as a subtype of atomic systems.
Name	Name of system.
SLName	Unmodified Simulink name. This is only written if it is *not* equal to Name.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`Identifier`	Unique identifier for this system.
`Hash`	Used in generating identifiers for systems during S-function importing.
`RTWSystemCode`	0: The system is inlined. When the system is inlined there is only one call site (i.e., CallSites has only one row). 1: The system is a function that accesses global data. In this mode, the system access state, block I/O, etc via global memory and there is only one call site (i.e, CallSites has only one row). 2: The system is a reused function where all state, block I/O, etc. is passed to the system via function arguments. A reused function can have multiple call sites.
`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.
`SystemIdx`	System index assigned to this system. Each system is assigned a unique non-negative integer by rtwgen. `CompiledModel.System[SystemIdx]` results in a reference to ourself.
`SL_SystemIdx`	This is for use by the Simulink Accelerator. It provides in index into Simulink's internal call graph structure for the systems.
`HStructDeclSystemIdx`	Where code, data structures, and local variables for this system are declared. • If `Inlined` == 0, then `DeclSystemIdx == SystemIdx` • If `Inlined == 1`, then we have either: **a** `DeclSystemIdx == SystemIdx`. This only occurs when this is a function-call subsystem and the caller is inlined, but placing the 'inlined' code with in a wrapper function. **b** `DeclSystemIdx` > `SystemIdx`. This occurs when a) does not. This gives an ancestor providing the location of where the various items are declared. For example this, will be the calling system (i.e. the parent) if it is not inlined, otherwise it will be the caller of the calling system (i.e. the grandparent) if it is not inlined, etc.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description

`CallSites`	*`[[callerSysIdx, callersCallSiteIdx, graphicalSysIdx, ssBlkIdxInGraphicalSys]; ...]`*
	An Mx4 matrix, enabling us to trace the call stack enabling us to generate fully qualified structure accesses. Each row lets us fully walk back the function call stack. For each row, we have:
	`  parentSys      = CompiledModel.System[CallSites[i,0]]`
	`  grandparentCSI = parentSys.CallSites[CallSites[i,1]]`
	`  graphicalSys   = CompiledModel.System[CallSites[i,2]]`
	`  subsysBlk      = graphicalSys.Block[CallSites[i,3]]`
	This is a recursive definition because we can apply `grandparentCSI` to get its parent call site, and so on. The `CallSites` information is needed to generate accesses to elements within the various structures (e.g. rtB).
	
	In general, *`callerSysIdx`* equals the *`graphicalSysIdx`* except for function-call subsystems crossing system boundaries.
	
	The number of rows (M) is the number of unique call stacks to this system (instances), i.e., how many times this system was reused. The `CallSites` allows us to answer "Who is calling us?"
	
	The `CallSites` can also be used to generate a list of names for a comment of who calls this function.
	
	`CallSites` also gives the number of instances of people calling us (i.e., different call stacks).

`NumChildSystems`	Number of systems that this system parents.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description

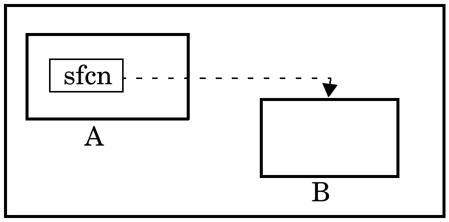
`ChildSystems`	A %<NumChildSystems> x 4 matrix where each row contains: *`[childSystemIdx, memoryReferenceCount, ... graphicalParentSysIdx, ssBlkIdxInGraphicalSys]`*. The *`memoryReferenceCount`* field is used in TLC when using the structure typedef's of the child systems in declaring the structure typedef for the current system. Nominally, the *`memoryReferenceCount`* will be 0. It will be non-zero for child function-call subsystems that are invoked from multiple sources (e.g., a wide function-call input port). For example, if a function-call subsystem is executed by three different initiators, then there will be three rows where that are identical, except for the *`memoryReferenceCount`* which takes on the values, 0, 1, 2. To illustrate the meaning of other fields, consider this model:  The resulting model.rtw will contain:
	```
	System { # B
	  NumChildSystems 0
	}
	System { # A
	  NumChildSystems 1
	  ChildSystems [0, 0, 2, 1]
	  ...
	  Block # sfcn (not accessable from ChildSystems)
	}
	System { # Root
	  NumChildSystems 1
	  ChildSystems [1, 0, 2, 0]
	  Block # Subsystem block corresponding to system A
	  Block # Subsystem block corresponding to system B
	}
	```

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`Interface {`	Always present, even if the system is inlined or it is the root system. This field contains information about the data structures that are contained in this system and how to generate the interface for the system. The interface record contains a number of argument definition records. For example, `CanonicalInputArgDef`. The presence of these records doesn't necessarily imply that the generated code will have these definitions as actual arguments to the code produced for this system. If the System is inlined or a non-reusable style function is being created, then the inputs are accessed via global data structures. If a reusable function is being generated for this system, then this arguments will only appear, if during code generation, blocks access the corresponding arguments. TLC tracks the access to the argument definitions, and after producing the 'guts' of the function, TLC wraps this code in a function prototype.
`NumCanonicalInputArgDef`	Length of the `CanonicalInputArgDef` records list giving the number of signals entering this system and are driven by blocks that are outside this system.
`CanonicaInputArgDef {`	
`Identifer`	String such as u0 giving the name to map any (u*i*) signal sources to.
`FirstSignalSrc`	Location of this signal in a specific data structure. It can be `B#` (`BlockOutputs`), `D#` (`DWork`), `X#` (`ContStates`) and the Width appended to it.e.g.,`B5@4` which refers to the `BlockOutput` record with index 5 and of width 4. This is used to get the data type info to generate the actual input argument, e.g., `real_T *signame`.
`FirstSignalOffset`	Used with `FirstSignalSrc` to find the corresponding offset for a signal.
`SignalWidth`	The record identified by `FirstSignalSrc` may be wider than the actual input signal. In this case, the call-site adds the appropriate offset (consider the case of a demux'd signal feeding a system).

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`}`	
`BlockIOArgDef {`	Definition for the block I/O argument that contains entries in the overall block I/O structure for this system. Note is possible that a system has no block I/O argument (e.g., a single scope block in a system) in which case we identify that the system requires no block I/O argument via during TLC execution. An 'is needed' flag tracked and set to true when a block within a system requires block I/O. Also consider a system that only contains a nonvirtual subsystem with no I/O and several blocks requiring block I/O. In this case `NumFlatFields` will be 0, but the lower system will require block I/O and this argument is required.
`FirstLocation`	Index into `CompiledModel.BlockOutputs.BlockOutput` record list giving the first instance starting location of the block outputs that belong to this system. This enables TLC to generate the procedure body when the system isn't inlined and there are multiple references because all the other references have the same pattern in the `BlockOutputs` structure. If the system is inlined then this is the only instance and we generate the inline code using the information from this index.
`NumFlatFields`	The `BlockOutputs` record list is sorted such the all `rtB_name` flat structure field members (e.g. `real_T speed[100]`) appear before any non-flat fields (e.g. `rtB_subname` subNameId). This can be zero if there are only substructures within `rtB_name`. Used to determine how many arguments come from the block I/O, i.e., there is one for each 'flat' field because these are referenced directly by blocks in this system.
`}`	
`NumCanonicalOutputArgDef`	Length of the `CanonicalOutputArgDef` record list giving the number of outputs in a system that don't come from the systems 'local block I/O'.
`CanonicalOutputArgDef {`	Name of identifier to map (`y`*`i`*) signal sources to. Note, in general most outputs become part of the systems `BlockIOArg`, except when a subsystem output feeds a merge block, in which case, canonical is passed out via an explicit function argument.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`Identifer`	"" if this is part of `BlockIOArg`, otherwise name for function prototype such as `y0`.
`FirstSignalSrc`	Location of this signal in a specific data structure. If the signal source has a width of `1`, `FirstSignalSrc` is formatted as `B1`. Otherwise it is formatted as `B`*`i`*`@`*`w`*, where *`i`* is the index into the `BlockOutput` record, and *`w`* is the signal width. For example, `B5@4` refers to a `BlockOutput` record with index `5` and of width `4`. See "Signal Connections" on page A-35 for a usage example.
`FirstSignalOffset`	Used with `FirstSignalSrc` to find the corresponding offset for a signal. The record identified by `FirstSignalSrc` may be wider than the actual output signal and because of the merge block capabilities, this output may be a subsection of the `FirstSignalSrc` record and this field gives the offset value
`SignalWidth`	`FirstSignalSrc` may be wider than the actual output signal and this gives the actual width of the signal. This happens when a output signal from a system feeds a merge block.
`}`	
`PrmArgDef {`	This contains the hierarchical parameter structure when inline parameters is off, and a semi-hierarchical parameter structure when inline parameters is on.
`FirstLocation`	
`NumFlatFields`	Similar to `BlockIOArgDef.FirstLocation`
`}`	Similar to `BlockIOArgDef.NumFlatFields`
`NumCanonicalPrmArgDefs`	Length of the `CanonicalPrmArg` records list.
`CanonicalPrmArgDef {`	One for each parameter argument. These originate when a variable in a parameter field (or child system Canonical parameter) maps back to a parameter field in a mask above us.
`Identifier`	Identifier name of the parameter.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description

`Dimensions`	`[1]   : A scalar`
	`[n]   : A vector`
	`[n,m] : A matrix`
	We use this to decide if we are passing by value or reference.

`DataTypeIdx`	Index into `CompiledModel.DataTypes.DataType`
`ComplexSignal`	`yes/no`: is this parameter argument complex?
`}`	
`DWorkArgDef {`	
`FirstLocation`	Index into `CompiledModel.DWorks.DWork` record list giving the starting location of the DWorks that belong to this system. Similar to `BlockIOArgDef.FirstLocation` handling.
`NumFlatFields`	Number of `DWork` records that belong to this system. Similar to `BlockIOArgDef.NumFlatFields` handling.
`}`	

`ContStatesArgDef {`	Used to generate the three input arguments to the system, *`rtX_name`*, *`rtXdot_name`*, *`rtXdis_name`* (disabled). We need to make sure that accesses to *`rtX_name`*, *`rtXdot`*, and no access to *`rtXdis_name`* result in only two arguments.
	
	Structures are generated via:
	`typedef struct rtX_name_tag {`
	`  real_T id1;`
	`  ...`
	`} rtX_name;`
	`typedef rtX_name rtXdot_name;`
	`typedef struct rtXdis_name_tag {`
	`  boolean_T id1;`
	`  ...`
	`} rtXdis_name;`

`FirstLocation`	Index into `CompiledModel.ContStates.ContState` record list giving the starting location of the `ContStates` that belong to this system. Similar to `BlockIOArgDef.FirstLocation` handling.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`NumFlatFields`	Number of `ContState` records that belong to this system. Similar to `BlockIOArgDef.NumFlatFields` handling
`}`	
`NonSampledZCArgDef {`	Generates arguments: `rtZC, rtZCdir`
`FirstLocation`	Index into `CompiledModel.NoncampledZCs.NonsampledZC` record list giving the starting location of the nonsampled zero crossings that belong to this system.
`NumFlatFields`	Number of `NonsampledZC` records that belong to this system.
`}`	
`ZCEventArgDef {`	Generates arguments: *`rtPZCE_name`*
`FirstLocation`	Index into `CompiledModel.ZCEvents.ZCEvent` record list giving the starting location of the ZC events that belong to this system.
`NumFlatFields`	Number of `ZCEvent` records that belong to this system. Similar to `BlockIOArgDef.NumFlatFields` handling.
`}`	End of `Interface` record.
`InEnableHierarchy`	Does this system reside within an enabled hierarchy (`yes`/`no`). This is used to aid in the generated code for the initialization of states, block I/O, etc.
`BlockControllingStateReset`	Optional for `InEnableHierarchy=1`, giving the *`[system, block]`* index for the parent enabled system that controls state reset for this system (which may be itself)
`NoCode`	If `yes`, generate no code. This system runs on the host only (during simulations or during external mode).
`NumChildPrmArgs`	Only present if inline parameters is on. Number of elements in an aggregated list of all the parameters passed to child systems. See subsystem `Block.CallSiteInfo.CanonicalPrmArg`
`ChildPrmArg {`	

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`TmpPrmId`	"" or id. If this is non-empty, then there are multiple references to this canonical parameter argument and for efficiency we should create a local variable assigned to the parameter expression.
`ASTNode {`	Handled just like a block `Parameter[i].ASTNode`
`}`	
`}`	
`NumZCEvents`	Number of zero-crossing events for all blocks in this system.
`SystemFileName`	Name of file into which generated code will be written. Simulink determines the appropriate filename based on Subsystem dialog settings and `AutoRTWFileName` so that TLC can directly use this file name. Only written for non-root systems.
`AutoRTWFileName`	yes/no: Flag indicating whether a system has its `RTWFileNameOpts` set to '`Auto`' or not. If set to '`Auto`', Simulink determines the filename into which the system will generate if it generates into a separate file.
`LibraryName`	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.
`StartFcn`	Name of start functions for nonvirtual subsystem.
`InitializeFcn`	Name of initialize function for enable systems that are configured to reset states.
`OutputFcn`	Name of output function for nonvirtual subsystem
`UpdateFcn`	Name of update function for nonvirtual subsystem.
`DerivativeFcn`	Name of derivative function for systems that have continuous states.
`EnableFcn`	Name of disable function for `enable` or `enable_with_trigger` systems.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`DisableFcn`	Name of disable function for `enable` or `enable_with_trigger` systems.
`ZeroCrossingFcn`	Name of nonsampled zero-crossing function for `enable` systems using variable step solver.
`OutputUpdateFcn`	Name of output/update function for `trigger` or `enable_with_trigger` systems.
`NumBlocks`	Number of nonvirtual blocks in the system.
`NumVirtualOutportBlocks`	For the root system, the number of virtual outport blocks is 0 (since all root outport blocks are nonvirtual). For a system corresponding to a conditionally executed subsystem, this is equal to the number of outport blocks in the subsystem. For each of these virtual outport blocks, there is a corresponding `Block` record which appears after all the nonvirtual `Block` records.
`VirtualOutportBlockIdx`	Starting index in the following `Block` record list of the virtual outport blocks.
`NumTotalBlocks`	Number of blocks in the system (sum of `NumBlocks` and `NumVirtualOutportBlocks`).
`Block {`	One for each nonvirtual block in the system. The virtual outport block records are described below.
`Type`	Block type, e.g., Gain.
`InMask`	`Yes` if this block *lives* 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.
`BlockIdx`	`[`*`systemIdx, callsiteIdx, blockIdx`*`]`
`ReducedBlocksConnected`	`Record of indices of connected blocks reduced`
`ExprCommentInfo {`	Record to hold block references of blocks folded into expressions. Record is used to create an appropriate comment.
`SysIdxList`	Initialized to []. Managed in TLC.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`  BlkIdxList`	Initialized to []. Managed in TLC.
`}`	
`ExprCommentSrcIdx`	Info for preserving comments
`  SysIdx`	Initialized to -1 set to at TLC runtime
`  BlkIdx`	Initialized to -1 set to at TLC runtime
`}`	
`Tag`	This is the text that can be attached to a block via the command: `set_param('block','Tag','text')` 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: `set_param('block','RTWData',val)` where `val` is a MATLAB struct of string. For example, `val.field1 = 'field1 value'` `val.field2 = 'field2 value'`
`  field1`	`field1 value`
`  field2`	`field2 value`
`}`	
`Name`	Block name preceded with a `<root>` or `<S#>` token.
`SLName`	Unmodified Simulink name. This is only written if it is *not* equal to `Name`.
`Identifier`	Unique identifer 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.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`OutputPortTIDs`	Only written if port sample time information is available.
`InputPortSampleTimes`	Only written if port sample time information is available.
`OutputPortSampleTimes`	Only written if port sample time information is available.
`InputPortOffsetTimes`	Only written if port sample time information is available.
`OutputPortOffsetTimes`	Only written if port sample time information is available.
`TID`	Task ID, which can be one of: • 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.
`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.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`ContStates`	`[width, rootContStatesIdx, rootDerivStateIdx]` `CompiledModel.ContStates.ContState[rootContStatesIdx]` record list gives us the corresponding entry in the root ContState record list for this block. Not present if no continuos states, `DefaultBlock.ContStates == [0, -1, -1]`.
`ModeVector`	`[width, firstRootDWorkIdx]`, where `CompiledModel.DWorks.DWork[firstRootDWorkIdx]` gives us the corresponding entry in the model-wide DWork record for this mode vector. Not present if no modes,
`RWork`	`[width, firstRootDWorkIdx]`, Similar to `ModeVector`
`IWork`	`[width, firstRootDWorkIdx]`, Similar to `ModeVector`
`PWork`	`[width, firstRootDWorkIdx]`, Similar to `ModeVector`
`DiscStates`	`[width, firstRootDWorkIdx]`, Similar to `ModeVector`
`NumDWork`	Number of `DWork` records a block has declared. There is one `DWork` record for each data type work vector in the block. The `DWork` records include the `ModeVector, RWork, IWork, PWork, DiscStates`, and explicitly defined `DWork` vectors.
`DWork {`	One record for each data type work vector.
`  Name`	A block defined name for the DWork vector. This defaults to DWORK# if not explicitly specified by the block.
`  FirstRootIdx`	`CompiledModel.DWorks.DWork[FirstRootIdx]` gives us the corresponding entry in the root `DWork` record list for this `DWork`.
`}`	
`NumNonsampledZCs`	Specified as [N,I], where N is the number of nonsampled zero-crossings and I is the index into the nonsampledZCs and nonsampled ZCdirs vectors.
`NonsampledZC {`	One record for each nonsampled zero-crossing. Used to generate/work with rtZC, rtZCdir data structures.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
FirstRootIdx	CompiledModel.NonsampledZCs.NonsampledZC[FirstRootIdx] gives us the first corresponding entry in the root Nonsampled record list for this nonsampled zero crossing.
MapIdx	The number of zero crossings a block may have is dependent on the type (e.g., sample time) of the signals entering the block's input port as well as things like the internal state of the block. For example, an absolute value block may have an input port width of 5, but only the 2nd and 4th elements of the input signal are continuous. The block has an overall sample time of continuous, but we only need track zero crossings on the 2nd and 4th elements. We could register 5 zero crossings, and the solver would be looking for zero crossings on all signals, but never see zero crossings on the 1st, 3rd, and 5th signals. To eliminate this inefficiency, we let blocks register the minimal number of zero and specify a local indices for each zero crossing giving information about which signals to return in the block's ZeroCrossing method. In our absolute value block example, we have two NonsampledZC records where MapIdx is 2 in the first record, and 4 in the second record.
Direction	Direction of zero-crossing: Falling, Any, Rising.
}	
ZCEvents	[width, rootZCEventIdx] where width is the number of zero-crossing events and CompiledModel.ZCEvents.ZCEvent[rootZCEventIdx] gives us the corresponding entry in the root ZCEvent record list for this ZCEvent zero crossing. The ZCEvent record list is used to generate rtPZCE (previous zero crossing event state).
ZCEvent {	One record for each zero-crossing event.
Type	Type of zero-crossing: DiscontinuityAtZC, ContinuityAtZC, TriggeredDisconAtZC.
Direction	Direction of zero-crossing: Falling, Any, Rising.
}	

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`RollRegions`	`RollRegions` is the contiguous regions defined by the inputs and *block width*. Block width is the overall width of a block after scalar expansion. `RollRegions` is provided for use by the `%roll` construct.
`NumDataInputPorts`	Number of data input ports. Only written if nonzero.
`DataInputPort {`	One record for each data input port.
`SignalSrc`	A vector of length `Width` where each element specifies the source index of a signal. This is an index into the `BlockOutput` record (`Bi`), an index into the `ContState` record (`Xi`), an index into the `DWork` record (`Di`), an index into the external input vector (`Ui`), unconnected ground (`G0`), or `F` indicating the source is a function-call. Within a non-inlined system this can be a local/relative `BlockOutput` record index (`bi`), local contstate (`xi`) or local `DWork` vector index (`di`), canonical input (`ui`) or canonical output (`yi`).NOTE: The width of the input port may not match the width found in the corresponding `BlockOutput` record. This occurs when using the `Merge` block because it is possible to merge small signals into large signals.
`SignalOffset`	A vector of length `Width`, where each element is an integer value giving the offset within the record specified by `SignalSrc`.
`FrameData`	yes/no: Is this port frame-based?
`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.
`SystemToCall`	[sysIdx, callSiteIdx]
`DataTypeIdx`	Index into the `CompiledModel.DataTypes.DataType` record list giving the data type of this port. Only written if not 0 (see `CompiledModel.DataInputPortDefaults.DataTypeIdx`).
`ComplexSignal`	Is this port complex? Only written if `yes`. The default from `CompiledModel.DataInputPortDefaults.ComplexSignal` is `no`.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`RecurseOnInput`	0/1: Do we nominally recurse on input signal (for expression folding)?
`SrcIdx`	Optional `[sysIdx, blkIdx, portIdx]` if `RecurseOnInput` is 1 (true).
`HaveGround`	yes/no: Is this port connected to ground?
`RollRegions`	A vector (e.g., [1:5, 6:10, 11]) giving the contiguous regions for this data input port over which *for* 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.
`SignalSrc`	Similar to `DataInputPort.SignalSrc`.
`SignalOffset`	Similar to `DataInputPort.SignalOffset`.
`Type`	Type of control port: `enable`, `trigger`, `ifaction` or `function-call.`
`SignalSrcTID`	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.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`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.
`Width` to `BufferDstPort`	Similar to the items in the `DataInputPort` record.
`}`	
`NumDataOutputPorts`	Number of output ports. Only written if nonzero.
`DataOutputPort {`	One record for each output port.
`SignalSrc`	Where this data output port writes to.A vector of length Width where each element specifies the source index of a `BlockOutput` record (`Bi`) or that output is a FcnCall (F). Within a non-inlined System that is going to be generated as a procedure, this can be a local/relative `BlockOutput` record index (`bi`), or canonical output (`yi`).
`SignalOffset`	A vector of length `Width`, where each element is an integer value giving the offset within the record specified by `SignalSrc`.
`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.
`ComplexSignal`	yes/no: Is this port complex?
`OutputExpression`	0/1, Is the output an expression (used by expression folding)?
`TrivialOutputExpression`	0/1, Is the output a trivial expression (used by expression folding)?
`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.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
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`.
InputPortConnected	Vector of length `NumDataInputPorts` containing `yes` or `no`
OutputPortConnected	Vector of length `NumDataOutputPorts` containing `yes` or `no`
InputPortBeingMerged	Vector of length `NumDataOutputPorts` containing `yes` or `no`
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 *connection points*. 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-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`DstBlockAndPortEl`	Vector of length four giving the destination connection: `[sysIdx, blkIdx, inputPortIdx, inputPortEl]` *sysIdx* is the index of the system record. *blkIdx* is the index with in system record of the destination block. *inputPortIdx* is the index of the destination input port. *inputPortEl* is the starting offset within the port of the connection.
`}`	
`}`	
`}`	
`ParamSettings {`	Optional record specific to block.
`blockSpecificName`	Block specific settings.
`}`	
`<S-function fields>`	Optional fields (parameters and/or records) that are written to the *`model`*`.rtw` file by the your specific S-function `mdlRTW` method.
`Parameters`	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`.
`Parameter {`	One record for each parameter.
`Name`	Name of the parameter as defined by the block.
`Dimensions`	Vector of the form `[`*`nRows, nCols`*`]` for the signal. Only written if number of dimensions is greater than 1.
`DataTypeIdx`	Data type index of the parameter into the `CompiledModel.DataTypes.DataType` records. Only written if not 0 (i.e., not `real_T`).
`ComplexSignal`	Is this parameter complex? Only written if `yes`.
`String`	String entered in the Simulink block dialog box.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`StringType`	One of: • `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`	• `Op = SL_CALCULATED => ASTNode` contains `ModelParametersIdx`, an index into the `ModelParameters` table for an evaluated (constant) parameter expression; • `Op = SL_NOT_INLINED => ASTNode` contains `ModelParametersIdx`, an index into the `ModelParameters` table for evaluated (calculated) parameter expressions when Inline Parameters is `off`; • `Op = SL_INLINED => ASTNode` contains `ModelParametersIdx`, an index into the `ModelParameters` table for an evaluated (constant) parameter expression when Inline Parameters is `on`; • `Op = M_ID` (a terminal node), the AST record contains ModelParameterIdx; • `Op = M_CANPRM_ID`, the AST record contains a `CanonicalPrmArgDefIdx` (an index that specifies which parameter argument of the system is used in the current parameter expression; • `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.
fields depend on Op	

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`}`	
`}`	
`ParamName``0`	Parameter[0] - An alias to the first parameter.
`...`	
`ParamName``N-1`	Parameter[N-1] - An alias to the last parameter.
`}`	
`CallSiteIdx`	The instance index of the block; only written for subsystem blocks
`CallSiteInfo {` `}`	Record containing the call-site information for Subsystem blocks. Only written for Subsystem block. See “Subsystem Block Specific Records” on page 112.
`HiddenDataInputPort {` `}`	Only written for nonvritual inport blocks within nonvirtual subsystems. The contents of this record are similar to `DataInputPort` records.
`HiddenControlInputPort {` `}`	Only written for enable and trigger port blocks. The contents of this record are similar to `ControlInputPort` records.
`Block {`	One block record (after the nonvritual block records) for each virtual outport 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 *not* equal to `Name`.
`Identifier`	Unique identifer across all blocks.
`RollRegions`	A vector (e.g., [1:5, 6:10, 11]) giving the contiguous regions over which *for* loops can be used.

Table A-24: Model.rtw System Record (Continued)

Variable/Record Name	Description
`  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.
`  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: `set_param('block','RTWData',val)` where val is a MATLAB struct of string. For example, `val.field1 = 'field1 value'` `val.field2 = 'field2 value'`
`    field1`	`field1 value`
`    field2`	`field2 value`
`}`	
`}`	

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-25: Model.rtw Stateflow Record

Variable/Record Name	Description
`SFLibraryNames {`	Only written if Stateflow charts exist in the model.
`  NumUniqueCharts`	Number of Chart records.
`  Chart {`	Record for a Stateflow library chart.
`    Name`	Name of the Stateflow library chart.
`    ReferencedBy`	An N-by-2 matrix. Each row specifies a system and block pair that identifies a instance of the library chart.
`  }`	
`}`	

Model Checksums

Checksums are created that are unique for each model.

Table A-26: Model.rtw Checksums

Variable/Record Name	Description
`BlockParamChecksum`	This is a hash-based checksum for the block parameter values and identifier names.
`ModelChecksum`	This is a hash-based checksum for the model structure.

Block Specific Records

Each block may have parameters. 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.

This section contains three tables documenting the following block-specific records:

- Table A-27 describes the block specific records written for the Simulink built-in blocks (excluding common fields described above). Each Simulink built-in block has an associated block record. The Target Language Compiler also has an associated TLC file for each block that specifies how code is generated for that block.
- Table A-28 describes the block specific records written for Subsystem blocks.
- Table A-29 describes the block specific records written for the Simulink linear blocks.

Table A-27: Model.rtw Block Specific Records

Block Type	Properties
`AbsoluteValue`	No block specific records
`Actuator`	No block specific records.
`ActionPort`	• `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.
`Assignment`	• `InIteratorSubsystem` - `0` or `1`. • `CopyInputToOutput` - `0` or `1`. Set to 1 if we need to copy input port to the outport. • `IteratorBlock` - Block index of parent iterator block. • `AssignMode` - mode that the block is in If the element, rows, or cols are internal, the following parameters are written: • `NumRows` • `NumCols` • `Elements` • `Rows` • `Columns` If the element, rows, or cols are not internal, the following parameters are written: • `ElementPortIdx` • `RowPortIdx` • `ColPortIdx`

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
`Backlash`	• `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. • `InputContiguous` (`yes` or `no`)
`BusSelector`	No block specific records.
`Clock`	No block specific records.
`CombinatorialLogic`	`TruthTable` parameter defining what the output should be, `y = f(TruthTable,u)`.
`ComplexToMagnitudeAngle`	`Output ParamSetting`. `Output` is `Magnitude`, `Angle`, or `Magnitude And Angle` indicating what the output port(s) are producing.
`ComplexToRealImag`	`Output ParamSetting`. `Output` is one of `Real`, `Imag`, or `Real And Imag` indicating what the output port(s) are producing.
`Constant`	`Value` parameter indicating what the output port should produce. `InFixptMode` (`1` or `0`)
`InFixPtMode`	Fixed point is output datatype (`1` or `0`).
`DataStoreMemory`	Virtual - Not written to the *`model`*`.rtw` file.
`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`	• `LowerValue` parameter - the lower value of the deadzone. • `UpperValue` parameter - the upper value of the deadzone. • `InputContiguous ParamSetting` (`yes` or `no`). • `OptimizeLowerLimit` (`yes` or `no`). • `SaturateOnOverflow ParamSetting` (`NotNeed`, `Needed`, `NeededBugOff`, or `NeededForDiagnostics`).
`Demux`	Virtual - Not written to the *`model`*`.rtw` file.

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	**Properties**
`Derivative`	The Derivative block computes its derivative by using the approximation: `(input-prevInput)/deltaT` 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: • `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 "Linear Block Specific Records" on page A-115.
`DiscreteIntegrator`	• Zero, one or two `RWork` vectors depending on the `InegratorMethod`. 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.

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
`DiscreteStateSpace`	See Model.rtw "Linear Block Specific Records" on page A-115
`DiscreteTransferFcn`	See Model.rtw "Linear Block Specific Records" on page A-115
`DiscreteZeroPole`	See Model.rtw "Linear Block Specific Records" on page A-115
`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`.
`EnablePort`	• `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.

`Fcn`	• `Expr ParamSetting` - Text string containing the expression the user entered.
	• `FcnASTNode` record, containing the parsed abstract syntax tree for the expression. The general form of the `ASTNode` is:
	`ASTNode {`
	`  Op          Operator (e.g. "+")`
	`  LHS {       Left-hand side argument for Op`
	`    ...`
	`  }`
	`  RHS {       Right-hand side argument for Op`
	`    ...`
	`  }`
	`}`

`ForIterator`	• `IterationSrc` - internal or external • `NumberOfIterations` -1 when external. Otherwise, the user specified parameter. • `OwnAssignmentBlocks` (1 or 0) • `HasParentIterator` (1 or 0) if this block is in a nested `for` or `while` loop, this is the topmost parent iterator block.
`From`	Virtual. Not written to *`model`*`.rtw` file.

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
`FromFile`	• `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. • One `PWork` vector, `PrevTimePtr` used for managing the block output.
`FromWorkspace`	• `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 “Using Simulink” manual). • `OutputAfterFinalValue ParamSetting` - How to generate output after final data value (see entry for From Workspace block in the “Using Simulink” manual). • `NumPoints ParamSetting` - Number of data points (rows) over which to read values from as time moves forward and write to the output port. 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`. • `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`, `RSimInfoPtr` used in computing the output.

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
Gain	• 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>]. • Gain.Parameter - Any real or complex value
Goto	Virtual. Not written to *model*.rtw file.
GotoTagVisibility	Virtual. Not written to *model*.rtw file.
Ground	Virtual. Not written to *model*.rtw 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).
HitCross	• InputContiguous ParamSetting - yes, no is the input contiguous? • HitCrossingOffset Parameter - The hit crossing offset used in computing the output.

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
If	• ElsePort - 0 or 1 (whether the else port is shown). • NumElseIfs - number of elseif ports • NumIfActionSystems - number of action subsystems connected to the If block. • TrueZCs - 0 or 1 (whether the If block is doing zero crossing on its input). • IfActionInfo - for each output port, a record of the conditional expression, and the subsystem that the port is connected to. • MinorStepGuard - yes, no. If yes, guard against changing output value during minor step.
InitialCondition	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 *model*.rtw file.
Integrator	• ExternalReset 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. • ExternalX0 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. • InitialCondition parameter. • UpperSaturationLimit parameter. • LowerSaturationLimit parameter.
Logic	Operator ParamSetting - one of AND, OR, NAND, NOR, XOR, or NOT.

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
`Lookup`	• `ZeroTechnique ParamSetting` - The type of lookup being performed. This doesn't change during model execution. The possibilities are: `NormalInterp`, `AverageValue`, or `Middle_Value.` • `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.
`Lookup2D`	• `ColZeroTechnique ParamSetting` - `NormalInterp`, `AverageValue`, or `MiddleValue`. • `ColZeroIndex ParamSetting` - Primary index when column data is zero. • `ColZeroIndex ParamSetting` - - `NormalInterp`, `AverageValue`, or `MiddleValue`. • `RowIndex` parameter - The *row* input values, `x`, to the function `z = f(x,y)`. • `ColumnIndex` parameter - The *column* input values, `y`, to the function `z = f(x,y)`. • `OutputValues` parameter - The *table* output values, `z`, for the function `z = f(x,y)`.
`MagnitudeAngleToComplex`	• `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`	• One `DWork` vector, `PreviousInput`, used to produce the output. • `X0` parameter - the initial condition.
`Merge`	`InitialOutput` parameter, giving the initial output for the merged signal.

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	**Properties**
`MinMax`	`Function ParamSetting - min` or `max`.
`MultiPortSwitch`	No block specific records.
`Mux`	Virtual. Not written to *`model`*`.rtw` file.
`Outport`	The block record for this block depends on the type of outport: • Root outports: ▪ `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: ▪ 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. • `VirtualizableRoot` - Can be virtual block at subsystem level (`yes` or `no`)
`Probe`	• `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)	• If block is configured for element-wise multiply, the block record contains: ▪ 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-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
`Product` (matrix multiply with one input port)	If block is configured for matrix multiply with one input port and block reduction is `off`, the block record contains: • `Multiplication ParamSetting - "Matrix(*)"` • `Inputs ParamSetting` - string vector of the form: `[*, *, /]` • `OneInputMultipley ParamSetting - yes.`

`Product` (matrix multiply with more than one input port).	If block is configured for matrix multiply, with more than one input port, the block record contains:
	• `Multiplication ParamSetting - "Matrix(*)"`
	• `Inputs ParamSetting` - string vector of the form: `[*, *, /]`
	• `OneInputMultipley ParamSetting - no.`
	• `OperandComplexity ParamSetting` - `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)`
	• `Dimensions ParamSetting` - `[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].
	• `Operands ParamSetting` - `[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).
	• `Complexities ParamSetting` - `[numSteps x 3]` matrix. Each row contains the {result, operand1, operand2}. Where:
	zero - real
	one - complex
	• `Operators ParamSetting` - `LU`, `Pivot`, `X dwork` indices.
	• `DivisionBuffers ParamSetting` - `LU`, `Pivot`, `X dwork` indices.

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
`PulseGenerator`	• `PhaseDelay ParamSetting`, giving the numerical phase delay. • `PulseType ParamSetting`, giving `Time-based` or `Sample-based`. • `PulseMode ParamSetting`, one of `Sample-based-Discrete`, `Time-based-Discrete`, or `Time-based-Variable`. The following depend on the `PulseMode`: ▪ For `Sample-based-Discrete and Time-based-Discrete` there is one `IWork` for `ClockTicksCounter`, used to manage the pulse. ▪ For `Time-based-Variable` (when using variable-step solver with rsim or Accelerator): - One `IWork` for `justEnabled`, used to indicate if the pulse has been re-enabled - One `IWork` for `currValue`, indicating the on/off status of the pulse - One `IWork` for `numCompleteCycles`, used to track how many cycles have been completed - One `RWork` for `nextTime`, used to track the time of next hit • `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.
`Quantizer`	`QuantizationInterval` parameter - numerical vector giving the quantization interval points.
`RandomNumber`	• One `IWork` vector `RandSeed`. • One `RWork` vector `NextOutput`. • `Mean` parameter - the mean of the random number generator. • `StdDev` parameter - the standard deviation of the random number generator.

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
`RateLimiter`	• 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`	• `Input ParamSetting - Real`, `Imag`, or `RealAndImag`. • `ConstantPart` parameter.
`Reference`	Will never appear in *`model`*`.rtw`.
`RelationalOperator`	• `Operator ParamSetting` - One of ==, ~=, <, <=, >=, >. • `InputContiguous ParamSetting` - `yes` or `no`. • `InFixptMode` - `1` or `0` If InFixptMode = 1, the following records appear: • `FixPtOutput` - `DataTypeId` • `FixPtRelOpStr` - operator string • `FixPt_u0Dominant` - `0`, `1`, or `-1`
`Relay`	• `InputContiguous ParamSetting` - `yes` or `no`. • `OnSwitchValue` parameter. • `OffSwitchValue` parameter. • `OnOutputValue` parameter. • `OffOutputValue` parameter.
`ResetIntegrator`	`InitialCondition parameter.`
`Rounding`	`Operator ParamSetting - floor`, `ceil`, `round`, or `fix`
`Saturate`	• `InputContiguous ParamSetting` - `yes` or `no`. • `UpperLimit` parameter. • `LowerLimit` parameter. • OptimizeLowerLimit - `yes` or `no` • OptimizeUpperLimit - `yes` or `no`

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
`Scope`	• 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`	When the Selector block is nonvirtual (in vector, internal elements mode), the following are written to the *`model`*`.rtw` file. `SelectorMode` - mode that the block is in If the element, rows, or cols are internal, one or more of the following parameters are written: • `NumRows` • `NumCols` • `Elements` • `Rows` • `Columns` If the element, rows, or cols are not internal, one or more of the following parameters are written: • `ElementPortIdx` • `RowPortIdx` • `ColPortIdx`

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
`S-Function`	The S-function has the following parameter settings: • `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 `ssGetUPtrs` (instead of `ssGetU`), then this `ParamSetting` 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 `RequiredContiguous` attribute for the corresponding input port to `true`. The Level 2 S-function will be using `ssGetInputPortSignal` (instead of `ssGetInputPortSignalPtrs`). • `InputContiguous` - 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 `mdlInitializeSizes` and `mdlInitializeSampleTimes` that get updated. The first column is the S-function sample time index, and the 2nd column is the corresponding `SampleTime` record of the model giving the `PeriodAndOffset`. For example, an inherited sample time will be assigned the appropriate sample time such as that of the driving block. In this case, the `SampleTimesToSet` will be `[0, <i>]` where `<i>` is the specific `SampleTime` record for the model.

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties

`S-Function (continued)`	• `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?
	 The next section contains information about function-call connections:
	 `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 *called*.
	 `BlockToCall` `[`*`systemIndex`*`, `*`blockIndex`*`]` or `unconnected`
	 `}`

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
`S-function (continued)`	If the S-function has a `mdlRTW` method, then additional items can be added. See *`matlabroot`*`/simulink/src/sfuntmpl_doc.c`. If the S-function is not inlined, i.e., *`sfunctionname.tlc`* does not exist) then 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. If the S-function is inlined, i.e., *`sfunctionname`*`.tlc` does exist, No `sizes` parameter. Parameter names are derived from the run-time parameter names.
`SignalGenerator`	• WaveForm ParamSetting - `sine`, `square`, or `sawtooth`. • TwoPi - 6.283185307179586. • `Amplitude` parameter. • `Frequency` parameter.
`Signum`	No block specific records.

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
`Sin`	• `- SineType ParamSetting`, indicating `Time-based` or `Sample-based`. • The remaining items depend on the mode of operation. If the `SineType` is `Time-based` and the block is discrete, the following are written to the *`model`*`.rtw` file: ▪ Two `RWork` vectors: `LastSin` and `LastCos`. ▪ One `IWork`: `SystemEnable`. ▪ Seven parameters: `Amplitude`, `Bias`, `Frequency`, `SinH`, `CosH`, `SinPhi`, and `CosPhi`. If the `SineType` is `Sample-based` and the block is discrete, the following are written to the *`model`*`.rtw` file: ▪ Two additional `ParamSettings`: `TsOffsetCorrection` and `Pi`. ▪ One `IWork`: `Counter`. ▪ Four parameters: `Amplitude`, `Bias`, `Samples` and `Offset`. If the SineType is `Time-based` and the block is continuous: ▪ Four parameters are written: `Amplitude`, `Bias`, `Frequency` and `Phase`.
`Step`	• `Time` parameter. • `Before` parameter. • `After` parameter.
`StateSpace`	See Model.rtw “Linear Block Specific Records” on page A-115

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
SwitchCase	• DefaultPort - 0 or 1 (whether the default port is shown). • NumCases - number of cases not including default case. • NumCaseActionSystems - number of action subsystems connected to the SwitchCase block. • TrueZCs - 0 or 1 (whether the SwitchCase block is doing zero crossing on its input). • CaseActionInfo - one record for each output port, consisting of. ▪ BlockToCall [systemIdx blockIdx] - the subsystem being called, or "unconnected" ▪ CaseConditions - integer vector containing conditions for each case; not present for default port. • MinorStepGuard - yes, no. If yes, guard against changing output value during minor step.
Sum	• Inputs ParamSetting - A vector of the form [+, +, -] corresponding to the configuration of the block. • SaturateOnOverflow ParamSetting (NotNeed, Needed, NeededBugOff, or NeededForDiagnostics).
SubSystem	Properties of the SubSystem record are described in "Subsystem Block Specific Records" on page A-112.
Switch	• ControlInputContiguous ParamSetting - yes or no. • Threshold parameter.
ToFile	• Two IWork vectors, Decimation, and Count. • Two PWork vectors, FilePtr, and LogFilePtr. • Filename ParamSetting. • MatrixName ParamSetting. • Decimation ParamSetting.

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
`ToWorkspace`	• One `PWork` vector, `LoggedData`. • `VariableName ParamSetting` - Name of variable used to save data. • `MaxDataPoints ParamSetting`- Maximum number of rows to save or 0 for no limit. • `Decimation ParamSetting` - Data logging interval • `InputContiguous` - yes or no. • `SaveFormat ParamSetting`. • `Label ParamSetting`.
`Terminator`	Virtual. Not written to *`model`*`.rtw` file.
`TransferFcn`	See Model.rtw "Linear Block Specific Records" on page A-115
`TransportDelay`	• `InitialInput ParamSetting`. • `BufferSize ParamSetting`. • `DiscreteInput ParamSetting`. • One `IWork` vector, `BufferIndices`. • One `PWork` vector, `TUbuffer`. • `DelayTime` parameter.
`TriggerPort`	• `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`	• `Operator ParamSetting` - `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `sinh`, `cosh`, or `tanh`.
`UniformRandomNumber`	• One `IWork` vector, `RandSeed`. • One `RWork` vector, `NextOutput`. • `Seed ParamSetting`. • `Minimum` parameter. • `MaxMinusMin` parameter.

Table A-27: Model.rtw Block Specific Records (Continued)

Block Type	Properties
UnitDelay	• One DWork vector, PreviousInput, called DSTATE, used to produce the output. • X0 parameter - the initial condition. • AccumType - Accumulator type, 0 (no accumulator), +, or - • UnityAccum - yes or no • SaturateAccumOnOverflow - NotNeeded or Needed When Fixed-point is in use, the following records appear: • InFixptMode - 1 • :FixPtOutput - DataTypeId
VariableTransportDelay	• InitialInput ParamSetting. • BufferSize ParamSetting. • DiscreteInput ParamSetting. • One RWork vector, TUbufferArea. • One PWork vector, TUbufferPtrs. • DelayTime parameter.
Width	No block specific records.
ZeroPole	See Model.rtw "Linear Block Specific Records" on page A-115
WhileIterator	• MaxNumberOfIterations - maximum number of time this loop will iterate. • OwnAssignBlocks - yes or no. yes if there are any Assignment blocks in the same system. • NeedIterationVariable - yes or no. • HasParentIterator - 0 or 1.
ZeroOrderHold	No block specific records.

Subsystem Block Specific Records

This table describes the block specific records written for the Simulink Subsystem block.

Table A-28: Model.rtw Subsystem Block Specific Records

`ParamSettings {`	Paramsettings for Subsystem Block
`SystemIdx`	Index of the `System` corresponding to this `Subsystem` block.
`StatesWhenEnabling`	`held` or `reset`. Only written if enable port is present,
`TriggerBlkIdx`	Block index of `TriggerPort` block in system or `-1` if the trigger block is not present
`IteratorBlock`	Block index of Iterator block, not present if there is no iterator.
`IteratorBlockType`	`for` or `while`: if one of them is present.
`IteratorScope`	Only written if the Subsystem has an iterator block. Can be `NoTIDScope` or `NeedTIDScope`.
`TriggerScope`	Only written if we have a trigger port. It can be one of: `NoScope`, `ScopeOnce`, `ScopeIndividually`.
`EnableScope`	Only written if we have an enable port. It can be one of: `NoScope`, `ScopeOnce`, `ScopeIndividually`
`EnableOutputScope`	Sample time scoping for enable/atomic system's Output code. It can be `yes` or `no`.
`SystemContStates`	Specified as `[N,I]` where `N` is the number of continuous states and `I` is the offset in the derivative vector.
`NumNonsampledZCs`	Number of NonsampledZCs in the subsystem.
`StartNonsampledZCs`	Start index of the NonsampledZCs corresponding to this system in `CompiledModel.NonsampledZCs.NonsampledZC`
`SingleRate`	Is this a SingleRate system. It can be `yes` or `no`.
`MinorStepGuard`	Does this system needs a minor step guard. It can be `yes` or `no`.
`}`	
`CallSiteInfo {`	This has information used in generating a call to this System in its caller or parent system and the typedef's need to execute the function (or inlined version of the function).
`SystemIdx`	Index of the corresponding called/inlined system.

Table A-28: Model.rtw Subsystem Block Specific Records (Continued)

`CallSiteIdx`	Used to determine the row in the `CallSites` matrix in the called/inlined system corresponding to this block. For example, let: `sys = CompiledModel.System[CallSiteInfo.SystemIdx]` then `cs = sys.CallSites[CallSiteIdx]` will point back to this block.
`StructId`	The identifier for the structure to be passed in (e.g. rtB_name id). The identifiers are unique at a given level. As an example, consider a model that has two references (sysA1,sysA2) to a nonvirtual subsystem named sysA. Within sysA, there are three references (sysB1,sysB2,sysB3) to sysB. Within sysB, there are five references (sysC1,sysC2,sysC3,sysC4,sysC5) to sysC. The total number of system instances: (see below)

```
  sysA: 2
  sysB: 2*3 = 6
  sysC: 2*3*5 = 30
```

The structure will look like:

```
  struct rtB_sysC_tag {
    <flat fields from regular block outputs in sysC>
} rtB_sysC;
struct rtB_sysB_tag {
  <flat fields from regular block outputs in sysB>
  rtB_sysC sysC1;
  rtB_sysC sysC2;
  rtB_sysC sysC3;
  rtB_sysC sysC4;
  rtB_sysC sysC5;
} rtB_sysB;
struct rtB_sysA_tag {
  <flat fields from regular block outputs in sysA>
  rtB_sysB sysB1;
  rtB_sysB sysB2;
  rtB_sysB sysB3;
} rtB_sysA;
struct rtB_tag {
  <flat fields from regular blocks at root>
  rtB_sysA sysA1;
  rtB_sysA sysA2;
}rtB;
```

This same `StructID` is used by the various Arg infos.

Table A-28: Model.rtw Subsystem Block Specific Records (Continued)

`NumCanonicalInputArgs`	Number of explicit inputs to the system.
`CanonicalInputArg {`	`One record for each explicit input.`
`  SignalSrc`	Vector specifying the input signal to the system. This can be global/local block I/O (`Bi` or `bi`), global/local continuous states (`Xi` or `xi`), global/local DWork vectors (`Di` or `di`), root or canonical inputs (`Ui` or `ui`), or ground connection (`G`).
`  SignalOffset`	Vector where each element is an integer value giving the offset within the record specified by `SignalSrc.`
`}`	
`BlockIOArg {`	Present if the corresponding system has block I/O.
`  OffsetIdx`	Integer index used with `Interface.BlockIOArgDef.FirstLocation` to get to the first `BlockOutput` slot corresponding to this subsystem, `i.e.,` `interface = System[CallSiteInfo.SystemIdx].Interface` `bioIdx = OffsetIdx + interface.BlockIOArgDef.FirstLocation`
`}`	
`NumCanonicalOutputArgs`	Number of explicit outputs from a Subsystem
`CanonicalOutputArg {`	One record for each explicit output.
`  SignalSrc`	Vector specifying the canonical output of the subsystem. This can be global/local block I/O (`Bi` or `bi`), root or canonical outputs of the parent system (`yi`).
`  SignalOffset`	Vector where each element is an integer value giving the offset within the record specified by SignalSrc.
`}`	
`PrmArg {`	
`  OffsetIdx`	Integer index used with `Interface.ParamArgDef.FirstLocation` to get to the first `BlockOutput` slot corresponding to this subsystem, `i.e.,` `interface= System[CallSiteInfo.SystemIdx].Interface` `prmIdx = OffsetIdx + interface.BlockIOArgDef.FirstLocation`
`}`	
`NumCanonicalPrmArgs`	Occur when inline parameters is on, i.e., always zero if Inline Parameters is off
`CanonicalPrmArg {`	One for each parameter argument. These originate when a variable in a block parameter field (or child system Canonical parameter) maps back to a parameter field in a mask above us.

Table A-28: Model.rtw Subsystem Block Specific Records (Continued)

`ChildPrmArgIdx`	Index in to the corresponding called/inlined system: `ChildCanoncialPrmArgs.ChildConicalPrmArg[i]`
`DWorkArg {`	Present if the corresponding system has DWorks.
`OffsetIdx`	Integer index used with `Interface.DWorkArgDef.FirstLocation` to get to the first DWork slot corresponding to this subsystem, `i.e.,` `interface = System[CallSiteInfo.SystemIdx].Interface` `dworkIdx = OffsetIdx + interface.DWorkArgDef.FirstLocation`
`}`	
`ContStatesArg {`	Present if the corresponding system has continuous states.
`OffsetIdx`	Integer index used with `Interface.ContStatesArgDef.FirstLocation` to get to the first continuous state corresponding to this subsystem. `i.e., interface= System[CallSiteInfo.SystemIdx].Interface` `csIdx=OffsetIdx+interface.ContStatesArgDef.FirstLocation`
`}`	
`NonsampledZCArg {`	Present if the corresponding system has nonsampled zero crossings.
`OffsetIdx`	Integer index used with `Interface.NonsampledZCArgsDef.FirstLocation` to get to the first DWork slot corresponding to this subsystem. `i.e., interface=System[CallSiteInfo.SystemIdx].Interface` `nsZCIdx = OffsetIdx +` `interface.NonsampledZCArgDef.FirstLocation`
`}`	
`ZCEventArg {`	Present if the corresponding system has ZC events.
`OffsetIdx`	Integer index used with `Interface.ZCEventArgDef.FirstLocation` to get to the first DWork slot corresponding to this subsystem. `i.e., interface= System[CallSiteInfo.SystemIdx].Interface` `dworkIdx = OffsetIdx +` `interface.ZCEventArgDef.FirstLocation`
`}`	

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-29: 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`
`  Value`	Vector that could be of zero length.
`  String`	`""`
`  StringType`	`Computed`
`}`	
`ParamSettings {`	

Table A-29: Model.rtw Linear Block Specific Records (Continued)

`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.
`}`	

Object Records

Several of the model records have `Object` sub-records that correspond to specific `Simulink.Signal` and `Simulink.Parameter` objects. These entities provide great flexibility in tailoring the generated code for custom environments by enabling you to associate TLC code with your own definitions for `Simulink.Signal` and `Simulink.Parameter` objects.

Table A-30: Model.rtw Object Record

Variable/Record Name	Description
`Object {`	The object record is a mapping between `Simulink.Signals` and `Simulink.Parameters`. Some common fields are show below. You can extend the output by adding information to particular `Simulink.Signal` and `Simulink.Parameter` objects.
`  Package`	`Simulink`.
`  Class {`	`Parameter` or `Signal`.
`    ObjectProperties {`	
`      RTWInfo {`	
`        Object {`	
`          Package`	`Simulink`
`          Class`	`RTWInfo`
`          ObjectProperites{`	
`            StorageClass`	`SimulinkGlobal or value specified by user.`
`            Alias`	`“Name”`
`          }`	
`      }`	

Table A-30: Model.rtw Object Record

Variable/Record Name	Description
`    }`	
`    Description`	“ “
`    DocUnits`	“ “
`  }`	
`}`	

TLC Error Handling

This chapter includes the following sections:

TLC Error Messages (p. B-5)	Use the `%exit` directive to generate errors from TLC files
TLC Function Library Error Messages (p. B-30)	Messages are sufficiently self-descriptive so that they do not need additional explanation

Generating Errors from TLC-Files

To generate errors from TLC files, use the `%exit` directive, but preferably one of the library functions described below that calls `%exit` for you. The two types of errors are

Usage errors	These can be caused by incorrect models.
Internal coding errors	These *cannot* be caused by incorrect models.

Usage Errors

Usage errors are errors resulting from incorrect models or attributes defined on a model. For example, suppose you have an S-Function block and an inline TLC file for a specific D/A device. If a model can contain only one copy of this S-function, then an error needs to be generated for a model that contains two copies of this S-Function block.

Using Library Functions

To generate usage errors related to a specific block, use the library function:

```
LibBlockReportError(block,"error string")
```

The `block` argument is the block record if it isn't scoped. If the block is currently scoped, then you can specify block as `[]`.

To generate general usage errors that are not related to a specific block, use

```
LibReportError("error string")
```

These library functions prepend the string `Real-Time Workshop Error` to the message you provide when reporting the error.

For an example usage of these functions, refer to `gensfun.tlc` for block errors and `commonsetup.tlc` for common errors. There are other files that use these functions in the TLC source directories within *`matlabroot`*`/rtw/c/tlc`.

Fatal (Internal) TLC Coding Errors

Suppose you have an S-function that has a local function that can accept only numerical numbers. You may want to add an *assert* requiring that the inputs be only numerical numbers. These asserts indicate fatal coding errors in that the user has no way of building a model or specifying attributes that can cause the error to occur.

Using Library Functions

The two available library functions are

```
LibBlockReportFatalError(block,"fatal coding error message")
```

where *block* is the offending block record (or `[]` if the block is already scoped), and

```
LibReportFatalError("fatal coding error message")
```

for error messages that are not block specific. For example, to add assert code you could use

```
%if TYPE(argument) != "Number"
   %<LibBlockReportFatalError(block,"unexpected argument type")
%endif
```

These library functions prepend the string `Real-Time Workshop Fatal` to the message you provide and display the call stack when reporting the error.

For an example usage of these functions, refer to `gensfun.tlc` for block errors and `commonsetup.tlc` for common errors. There are other files that use these functions in the directory `matlabroot/rtw/c/tlc`.

Using %exit

You can call `%exit` to generate fatal error messages, however, it is suggested that you use one of the previously discussed library functions. If you do use `%exit`, take care when generating an error string containing new lines (carriage returns); see Formatting Error Messages.

When generating fatal error messages directly with `%exit`, it is good practice to give a stack trace with the error message. This lets you see the call chain of functions that caused the error. To generate a stack trace, generate the message using the format

```
%setcommandswitch "-v1"
%exit RTW Fatal: error string
```

Formatting Error Messages

You should be careful when formatting error message strings. For example, suppose you create a local variable (called `message`) that contains text that has new lines.

```
%openfile message
My message text
with new lines (carriage returns)
%closefile message
```

If you then want to create another variable and prefix this message with the text "RTW Error:", you need to use

```
%openfile errorMessage
RTW Error: %<message>
%closefile errorMessage
```

or

```
%assign errorMessage = "RTW Error:"+ message
```

The statement

```
%assign errorMessage = "RTW Error: %<message>"
```

will cause a syntax error during TLC execution and your message will not be displayed. This should be avoided. Use the function `LibBlockReportError` to help prevent this type of runtime syntax error. The syntax error occurs because TLC evaluates the message, which causes new lines to appear in the assignment statement that appear as unterminated text strings (i.e., the trailing quote is missing).

After formatting your error message, use `LibBlockReportError`, a similar function, or `%exit` to report your error when it occurs.

Testing Error Messages

It is strongly suggested that you test your error messages before releasing your new TLC code. To test your error messages, copy the relevant code into a `test.tlc` file and run

```
tlc test.tlc
```

at the MATLAB prompt.

TLC Error Messages

This section lists and describes error messages generated by the Target Language Compiler (`tlc.mex`). Use this reference to

- Confirm that an error has been reported.
- Determine possible causes for an error.
- Determine possible ways to correct an error.

%closefile or %selectfile or %flushfile argument must be a valid open file

When using `%closefile` or `%selectfile` or `%flushfile`, the argument must be a valid file variable opened with `%openfile`.

%define no longer supported, use %function instead

Macros are no longer supported. You must rewrite all macros as functions or inline them in your code.

%error directive: *text*

Code containing the `%error` directive generates this message. It normally indicates some condition that the code was unable to handle and displays the text following the `%error` directive.

%exit directive: *text*

Code containing the `%exit` directive causes this message. It typically indicates some condition that the code was unable to handle and displays the text following the `%exit` directive. Note that this directive causes the Target Language Compiler to terminate regardless of the `-mnumber` command line option.

%filescope has already been used in this file.

The user attempted to use the `%filescope` directive more than once in a file.

%trace directive: *text*

The `%trace` directive produces this error message and displays the text following the `%trace` directive. Trace directives are only reported when the `-v` option (verbose mode) appears on the command line. Note that `%trace` directives are not considered errors and do not cause the Target Language Compiler to stop processing.

%warning directive: %s

The `%warning` directive produces this error message and displays the text following the `%warning` directive. Note that `%warning` directives are not considered errors and do not cause the Target Language Compiler to stop processing.

A %implements directive must appear within a block template file and must match the %language and type specified

A block template file was found, but it did not contain a `%implements` directive. A `%implements` directive is required to ensure that the correct language and type are implemented by this block template file. See "Object-Oriented Facility for Generating Target Code" on page 6-34 for more information.

A %switch statement can only have one %default

The user has written a `%switch` statement with multiple `%default` cases, as in the following example:

```
%switch expr
  %case 1
    code...
    %break
  %default

more code...
    %break
  %default  %% error
    even more code...
    %break
%endswitch
```

A language choice must be made using the %language directive prior to using GENERATE or GENERATE_TYPE

To use the `GENERATE` or `GENERATE_TYPE` built-in functions, the Target Language Compiler requires that you first specify the language being generated. It does this to ensure that the block-level target file implements the same language and type as specified in the `%language` directive.

A non-homogenous vector was passed to GENERATE_FORMATTED_VALUE

The builtin `GENERATE_FORMATTED_VALUE` can only process vectors which have homogenous elements (that is, vectors in which all the elements have the same type).

Ambiguous reference to *identifier* — must use array index to refer to one of multiple scopes

When using a repeated scope identifier from a database file, you must specify an index in order to disambiguate the reference. For example:

```
Database file:
block
{
   Name             "Abc2"
   Parameter {
       Name         "foo"
       Value        2
   }
}
block
{
   Name             "Abc3"
   Parameter {
       Name         "foo"
       Value        3
   }
}

TLC file:
%assign y = block
```

In this example, the reference to block is ambiguous because multiple repeated scopes named “block” appear in the database file. Use an index to disambiguate it, as in

```
%assign y = block[0]
```

An %if statement can only have one %else

The user has written an `%if` statement with multiple `%else` blocks, as in the following example.

```
%if expr
  code...
%else
  more code...
%else          %% error
  even mode code...
%endif
```

Argument to *identifier* must be a string

The following built-in functions expect a string and report this error if the argument passed is not a string.

CAST	GENERATE_FILENAME
EXISTS	GENERATE_FUNCTION_EXISTS
FEVAL	GENERATE_TYPE
FILE_EXISTS	GET_COMMAND _SWITCH
FORMAT	IDNUM
GENERATE	SYSNAME

Arguments to *directive* must be records

Arguments to `%mergerecord` and `%copyrecord` must be records. Also, the first argument to the following builtins must be records:

- `ISALIAS`
- `REMOVEFIELD`
- `FIELDNAMES`
- `ISFIELD`
- `GETFIELD`
- `SETFIELD`.

Arguments to TLC from the MATLAB command line must be strings

An attempt was made to invoke the Target Language Compiler from MATLAB and some of the arguments that were passed were not strings.

Assertion failed

An expression in an `%assert` statement evaluated to false.

Assignment to scope *identifier* is only allowed when using the + operator to add members

Scope assignment must be `scope = scope + variable`.

Attempt to define a function *identifier* on top of an existing variable or function

A function cannot be defined twice. Make sure that you don't have the same function defined in separate TLC files.

Attempt to divide by zero

The Target Language Compiler does not allow division by zero.

Bad cast - unable to cast this expression to "*type*"

The Target Language Compiler does not know how to cast this expression from its current type to the specified type. For example, the Target Language Compiler is not able to cast a string to a number as in

```
%assign x = "1234"
%assign y = CAST("Number", x );
```

Bad directory (*dirname*) in -O: *filename*

The `-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. For example:

```
%<foo(%<expr>)>
```

Instead, do the following:

```
%assign tmp = %<expr>
%<foo(tmp)>
```

Expansion directives %<> cannot span multiple lines; use \ at end of line

An expansion directive cannot span multiple lines. To work around this restriction, use the \ line continuation character. For example,

```
%<CompiledModel.System[Sysidx].Block[BlkIdx].Name +
"Hello">
```

is illegal, whereas

```
%<CompiledModel.System[Sysidx].Block[BlkIdx].Name + \
"Hello">
```

is correct.

Extra arguments to the *function-name* built-in function were ignored (Warning)

The following built-in functions report this warning when too many arguments are passed to them.

`CAST`	`NUMTLCFILES`
`EXISTS`	`OUTPUT_LINES`
`FILE_EXISTS`	`SIZE`
`FORMAT`	`STRING`
`GENERATE_FILENAME`	`STRINGOF`
`GENERATE_FUNCTION_EXISTS`	`SYSNAME`
`IDNUM`	`TLCFILES`
`ISFINITE`	`TYPE`
`ISINF`	`WHITE_SPACE`
`ISNAN`	`WILL_ROLL`

File name too long (directory = 'dirname', name = 'filename')

The specified filename was too long. The default limits are 256 characters for filename and 1024 characters for pathname, but the limits may be larger depending on the platform.

format is not a legal format value

The specified format was not legal for the `%realformat` directive. Valid format strings are `"EXPONENTIAL"` and `"CONCISE"`.

Function argument mismatch; function *function_name* expects *number* arguments

When calling a function, too many or too few arguments were passed to it.

Function reached the end and did not return a value

Functions that are not declared as `void` or `Output` must return a value. If a return value is not desired, declare the function as `void`, otherwise ensure that it always returns a value.

Function values are not allowed

Attempt to use a TLC function as a variable.

Identifier *identifier* multiply defined. Second and succeeding definitions ignored.

The user is attempting to add the same field to a record more than once, as in the following code.

```
%createrecord err { foo 1; rec { val 2 } }
%addtorecord err foo 2            %% error
```

Identifier *identifier* used on a %foreach statement was already in scope (Warning)

The argument to a `%foreach` statement cannot be defined prior to entering the `%foreach`.

Illegal use of eval (i.e. %<...>)

It is illegal to use evals in `.rtw` files. There are also some places where evals are not allowed in directives, for example

```
%function %<foo>(a, b, c) void  %% error
%endfunction
```

Indices may not be negative

An index used in a `[]` expression must be a nonnegative integer.

Indices must be constant integral numbers

An index used in a `[]` expression must be an integral number.

Invalid handle

An invalid handle was passed to the Target Language Compiler Server Mode.

Invalid identifier range, the leading strings *string1* and *string2* must match

When using a range of signals, for example, `u1:u10`, the identifier in the first argument did not match the identifier in the second.

Invalid identifier range, the lower bound (%d) must be less than the upper bound (%d)

When using a range of signals, for example, `u1:u10`, the lower bound was higher than the upper bound.

Invalid type for unary *operator*

Unary operators `-` and `+` require numeric types. Unary operator `~` requires an integral type. Unary operator `!` requires a numeric type.

Invalid type *type*

An invalid type was passed to a built-in function.

It is illegal to return a function from a function

A function value cannot be returned from a function call.

Named value *identifier* already exists within this *scope-identifier*; use %assign to change the value

You cannot use the block addition operator + to add a value that is already a member of the indicated block. Use `%assign` to change the value of an existing value. This example produces this error:

```
%assign x = BLK { a 1; b 2 }
%assign a = 3
%assign x = x + a
```

Use this instead:

```
%assign x.a = 3
```

No %case statement(s) seen yet, statement ignored.

Statements that appear inside a `%switch` statement, but precede any `%case` statements, are ignored, as in the following code.

```
%switch expr
```

```
%assign x = 2  %% this statement will be ignored
  %case 1
    code
    %break
%endswitch
```

Only double and character arrays can be converted from MATLAB to TLC. This can occur if the MATLAB function does not return a value (see %matlab).

Only double and character arrays can be converted from MATLAB to the Target Language Compiler. This error can occur if the MATLAB function does not return a value (see `%matlab`). For example:

```
%assign a = FEVAL(“int8“,3)
%matlab disp(a)
```

Only one output is allowed from the TLC

An attempt was made to receive multiple outputs from the MATLAB version of the Target Language Compiler.

Only strings of length 1 can be assigned using the [] notation

The right-hand side of a string assignment using the `[]` operator must be a string of length 1. You can only replace a single character using this notation.

Only strings or cells of strings may be used as the argument to Query and ExecString

A cell containing nonstring data was passed as the third argument to `Query` or `ExecString` in Server Mode.

Only vectors of the same length as the existing vector value can be assigned using the [] notation

When using the `[]` notation to replace a row of a matrix, the row must be a vector of the same length as the existing rows.

Output file *identifier* opened with %openfile was not closed

Output files opened with `%openfile` must be closed with `%closefile`. `identifier` is the name of the variable specified in the `%openfile` directive.

Note This might also occur if there is a syntax error in your code section between an `openfile` and `closefile`, or if you try to assign the output of a function of type `void` or `Output` to a variable.

Ranges, identifier ranges, and repeat values cannot be repeated

You cannot repeat a range, idrange, or repeat value. This prevents things like `[1@2@3]`.

String cannot modify the setting for the command line switch '*-switch*'

`%setcommandswitch` does not recognize the specified switch, or cannot modify it (e.g., `-r` cannot be modified).

'*String*' is not a recognized user defined property of this handle

The query performed on a TLC server mode handle is looking for an undefined property.

Syntax error

The indicated line contains a syntax error, See Chapter 6, "Directives and Built-in Functions," for information on the syntax.

The %break directive can only appear within a %foreach, %for, %roll, or %switch statement

The `%break` directive can only be used in a `%foreach`, `%for`, `%roll`, or `%switch` statement.

The %case and %default directives can only be used within the %switch statement

A `%case` or `%default` directive can only appear within a `%switch` statement.

The %continue directive can only appear within a %foreach, %for, or %roll statement

The `%continue` directive can only be used in a `%foreach`, `%for`, or `%roll` statement.

The %foreach statement expects a constant numeric argument

The argument of a `%foreach` must be a numeric type. For example,

```
%foreach Index = [1 2 3 4]
…
%endforeach
```

`%foreach` cannot accept a vector as input.

The %if statement expects a constant numeric argument

The argument of a `%if` must be a numeric type. For example:

```
%if [ 1 2 3 ]
…
%endif
```

`%if` cannot accept a vector as input.

The %implements directive expects a string or string vector as the list of languages

You can use the `%implements` directive to specify a string for the language being implemented, or to indicate that it implements multiple languages by using a vector of strings. You cannot specify any other argument type to the `%implements` directive.

The %implements directive specifies *type* as the type where *type* was expected

The type specified in the `%implements` directive must exactly match the type specified in the block or on the `GENERATE_TYPE` directive. If you want to specify that the block accept multiple input types, use the `%implements *` directive, as in

```
%implements * "C"   %% I accept any type and generate C code
```

The %implements language does not match the language currently being generated (*language*)

The language or languages specified in the `%implements` directive must exactly match the `%language` directive.

The %return statement can only appear within the body of a function

A `%return` statement can only be in the body of a function.

The == and != operators can only be used to compare values of the same type

The `==` and `!=` operator arguments must be the same type. You can use the `CAST()` built-in function to change them into the same type.

The argument for %openfile must be a valid string

When opening an output file, the name of the file must be a valid string.

The argument for %with must be a valid scope

The argument to `%with` must be a valid scope identifier. For example:

```
%assign x = 1
%with x
…
%endwith
```

In this code, the `%with` statement argument is a number and produces this error message.

The argument for an [] operation must be a repeated scope symbol, a vector, or a matrix

When using the `[]` operator to index, the expression on the left of the brackets must be a vector, matrix, string, numeric constant, or a repeated scope identifier. When using array indexing on a scalar, the constant is automatically scalar expanded and the value of the scalar is returned. For example:,

```
%openfile x
%assign y = x[0]
```

This example would cause this error because `x` is a file and is not valid for indexing.

The argument to %addincludepath must be a valid string

The argument to `%addincludepath` must be a string.

The argument to %include must be a valid string

The argument to the input file control directive must be a valid string with the filename given in double quotes.

The *begin* directive must be in the same file as the corresponding *end* directive.

These Target Language Compiler `begin` directives must appear in the same file as their corresponding `end` directives: `%function`, `%switch`, `%foreach`, `%roll`, and `%for`. Place the construct entirely within one Target Language Compiler source file.

The *begin* directive on this line has no matching *end* directive

For block-scoped directives, this error is produced if there is no matching `end` directive. This error can occur for the following block-scoped Target Language Compiler directives.

Begin Directive	End Directive	Description
`%if`	`%endif`	Conditional inclusion
`%for`	`%endfor`	Looping
`%foreach`	`%endforeach`	Looping
`%roll`	`%endroll`	Loop rolling
`%with`	`%endwith`	Scoping directive
`%switch`	`%endswitch`	Switch directive
`%function`	`%endfunction`	Function declaration directive
`{`	`}`	Record creation

The error is reported on the line that opens the scope and has no matching `end` scope.

Note Nested scopes must be closed before their parent scopes. Failure to include an `end` for a nested scope often causes this error, as in

```
%if Block.Name == "Sin 3"
    %foreach idx = Block.Width
%endif  %% Error reported here that the %foreach was not terminated
```

The construct %matlab *function_name*(...) construct is illegal in standalone tlc

You cannot call MATLAB from stand-alone TLC.

The FEVAL() function can accept only 2-dimensional arrays from MATLAB, not *number* dimensions

Return values from MATLAB can have at most two dimensions.

The FEVAL() function can accept vectors of numbers or strings only when calling MATLAB

Vectors passed to MATLAB can be numbers or strings. See “FEVAL Function” on page 6-46.

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 or complex valued argument

These functions expect a `Real` or complex value as the input argument.

The language being implemented cannot be changed within a block template file

You cannot change the language using the `%language` directive within a block template file.

The language being implemented has changed from *old-language* to *new-language* (Warning)

The language being implemented should not be changed in midstream because `GENERATE` function calls that appear prior to the `%language` directive may cause `generate` functions to load for the prior language. Only one language directive should appear in a given file.

The left-hand side of a . operator must be a valid scope identifier

When using the `.` operator, the left-hand side of the `.` operator must be a valid in-scope identifier. For example,

```
%assign x = 1
%assign y = x.y
```

In this code, the reference to `x.y` produces this error message because `x` is not defined as a scope.

The left-hand side of an assignment must be a simple expression comprised of ., [], and identifiers

Illegal left-hand side of assignment.

The number of columns specified (*specified-columns*) did not match the actual number of columns in all of the rows (*actual-columns*)

When specifying a Target Language Compiler matrix, the number of columns specified did not match the actual number of columns in the matrix. For example,

```
%assign mat = Matrix(2,1) [[1,2];[2,3]]
```

In this case, the number of columns in the declaration of the matrix (1) did not match the number of columns seen in the matrix (2). Either change the number of columns in the matrix, or change the matrix declaration.

The number of rows specified (*specified-rows*) did not match the actual number of rows seen in the matrix (*actual-rows*)

When specifying a Target Language Compiler matrix, the number of rows specified did not match the actual number of rows in the matrix. For example,

```
%assign mat = Matrix(1,2) [[1,2];[2,3]]
```

In this case, the number of rows in the declaration of the matrix (1) did not match the number of rows seen in the matrix (2). Either change the number of rows in the matrix or change the matrix declaration.

The *operator_name* operator only works on Boolean arguments

The `&&` and `||` operators work on Boolean values only.

The *operator_name* operator only works on integral arguments

The `&`, `^`, `|`, `<<`, `>>` and `%` operators only work on numbers.

The *operator_name* operator only works on numeric arguments

The arguments to the following operators both must be either Number or Real: `<`, `<=`, `>`, `>=`, `-`, `*`, `/`. This can also happen when using + as an unary operator. In addition, the `FORMAT` built-in function expects either a Number or Real argument.

The return value from the RollHeader function must be a string

When using `%roll`, the `RollHeader()` function specified in `Roller.tlc` must return a string value. See `%roll` for a complete discussion of the `%roll` construct.

The roll argument to %roll must be a nonempty vector of numbers or ranges

When using `%roll`, the `roll` vector cannot be empty and must contain numbers or ranges of numbers. See `%roll` for a complete discussion of the `%roll` construct.

The second value in a Range must be greater than the first value

When using a range, for example, `1:10`, the lower bound was higher than the upper bound.

The specified index (*index*) was out of the range 0 - *number-of-elements – 1*

This error occurs when indexing into any nonscalar beyond the end of the variable. For example:

```
%assign x = [1 2 3]
%assign y = x[3]
```

This example would cause this error. Remember, in the Target Language Compiler, array indices start at 0 and go to the number of elements minus 1.

The STRINGOF built-in function expects a vector of numbers as its argument

The `STRINGOF` function expects a vector of numbers. The function treats each number as the ASCII value of a valid character.

The SYSNAME built-in function expects an input string of the form <xxx>/yyy

The `SYSNAME` function takes a single string of the form `<xxx>/yyy` as it appears in the `.rtw` file and returns a vector of two strings `xxx` and `yyy`. If the input argument does not match this format, it returns this error.

The threshold on a %roll statement must be a single number

When using `%roll`, the roll threshold specified must be a single number. See `%roll` for a complete discussion of the `%roll` construct.

The use of *feature* is being deprecated and will not be supported in future versions of TLC. See the TLC manual for alternatives.

The `%define` and `%generate` directives are not recommended, as they are being replaced.

The WILL_ROLL built in function expects a range vector and an integer threshold

The `WILL_ROLL` function expects two arguments: a range vector and a threshold.

There are no more free contexts. Use tlc('close', HANDLE) to free up a context

The global context table has filled up while using the TLC server mode.

There was no type associated with the given block for GENERATE

The scope specified to `GENERATE` must include a `Type` parameter that indicates which template file should be used to generate code for the specified scope. For example,

```
%assign scope = block { Name "foo" }
%<GENERATE( scope, "Output" )>
```

This example produces the error message because the scope does not include the parameter `Type`. See the `GENERATE` and `GENERATE_TYPE` functions for more information and examples on using the `GENERATE` built-in function.

This assignment would overwrite an identifier-value pair from the RTW file. To avoid this error either qualify the left-hand side, or choose another identifier.

The user is trying to modify a field of a record in a `%with` block without qualifying the left-hand side, as in this example:

```
%createrecord foo { field 1 }
%with foo
  %assign field = 2  %% error
%endwith
```

The correct method is:

```
%createrecord foo { field 1 }
  %with foo
    %assign foo.field = 2
  %endwith
```

TLC has leaked *number* symbols. You may have created a cyclic record. If this not the case then please report this leak to The MathWorks.

There has been a memory leak while running TLC. The most common cause of this is having cyclic records.

Unable to find *identifier* within the *scope-identifier* scope

The given identifier was not found in the scope specified. For example:

```
%assign scope = ascope { x 5 }
%assign y = scope.y
```

In this code, the reference to `scope.y` produces this error message.

Unable to open %include file *filename*

The file included in a `%include` directive was not found on the path. Either locate the file and use the `-I` command line option to specify the correct directory, or move the file to a location on the current path.

Unable to open block template file *filename* from GENERATE or GENERATE_TYPE

When using `GENERATE`, the given filename was not found on the Target Language Compiler path. You can

- Add the file into a directory on the path.
- Use the `%generatefile` directive to specify an alternative filename for this block type that is on the path.
- Add the directory in which this file appears to the command line options using the `-I` switch.

Unable to open output file *filename*

Unable to open the specified output file; either an invalid filename was specified or the file was read only.

Undefined identifier *identifier_name*

The identifier specified in this expression was undefined.

Unknown type "*type*" in CAST expression

When calling the `CAST` built-in function, the type must be one of the valid Target Language Compiler types found in the Target Language Values table.

Unrecognized command line switch passed to *string*: *switch*

When querying the current state of a switch, the switch specified was not recognized.

Unrecognized directive "*directive-name*" seen

An illegal `%` directive was encountered. The valid directives are shown below.

`%addincludepath`	`%filescope`
`%addtorecord`	`%for`
`%assert`	`%foreach`
`%assign`	`%function`
`%break`	`%generate`

`%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 multiline comment.

A multiline (i.e. /% %/) comment has no terminator, as in the following code:

```
/% my comment

%assign x = 2
%assign y = x * 7
```

Unterminated string

A string must be closed prior to the end of an expansion directive or the end of a line.

Usage: tlc [options] file

Message	Description
`-r <name>`	Specify the Real-Time Workshop file to read.
`-v[<number>]`	Specify the verbose level to be <number> (1 by default).
`-I<path>`	Specify a path to local include files. The TLC will search this path in the order specified.
`-m[<number>|a]`	Specify the maximum number of errors (`a` is all). Default is 5.
`-O<path>`	Specify the path used to create output files. By default all TLC output will be created in this directory.

Message	Description
`-d[a|c|n|o]`	Invoke the TLC debug mode. `-da` will make TLC execute any %assert directives. `-dc` will invoke TLC command line debugger. `-dn` will cause TLC to produce log files indicating which lines were and were not hit during compilation. `-do` will disable TLC debugging behavior.
`-a<ident>=<expression>`	Assign a variable to a specified value. Use this option to specify parameters that can be used to change the behavior of your TLC program. This option is used by Real-Time Workshop to set options like inlining of parameters, file size limits, etc.
`-p<number>`	Print a '.' indicating progress for every <number> of TLC primitive operations executed.
`-lint`	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

```
%<Function>
```

call it with the appropriate arguments.

Values of type Special, Macro Expansion, Function, File, Full Identifier, and Index cannot be converted to MATLAB variables

The specified type cannot be converted to MATLAB variables.

When appending to a buffer stream, the variable must be a string

You can specify the `append` option for a buffer stream only if the variable currently exists as a string. Do not use the `append` option if the variable does not exist or is not a string. This example produces this error.

```
%assign x = 1
%openfile x , "a"
%closefile x
```

TLC Function Library Error Messages

There are many error messages generated by the TLC function library that are not documented. These messages are sufficiently self-descriptive so that they do not need additional explanation. However, if you come across an error message that you feel needs more description, contact our technical support staff and we will update it in a future release (and give more explanation).

Using TLC with Emacs

This chapter includes the following section:

The Emacs Editor (p. C-2)	Use the Emacs editor to edit your TLC files

The Emacs Editor

If you're editing TLC files, we recommend trying to use Emacs. You can get a copy of Emacs from `http://www.gnu.org`.

The MathWorks has created a `tlc-mode` for Emacs that gives automatic indenting and color-coded syntax highlighting of TLC files. You can obtain `tlc-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 `tlc-mode`.

Color-coding syntax in Emacs makes TLC code is much more readable.

Getting Started

To get started using Emacs:

Ctrl+x Ctrl+f `file.tlc <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 *.tlc
```

in the Unix directory where your `.tlc` files are located. The `etags` command is located the `emacs_root/bin` directory. Users of Windows NT must type

```
etags "--regex=/[ \t]*\%function[ \t]+.+/" --language=none *.tlc
```

in a DOS command window.

Index

Symbols
! 6-23
- 6-23
– 6-24
!= 6-25
% 6-2, 6-21, 6-23
& 6-25
&& 6-25
() 6-23
* 6-23
+ 6-23, 6-24
, 6-26
. 6-23
... 6-18
.c file 1-5
.h file 1-5
.log 7-10
.rtw file 1-5
/ 6-23
:: 6-22, 6-52
< 6-25
<< 6-24
<= 6-25
== 6-25
> 6-24
>= 6-25
>> 6-24
? : 6-26
\ 6-18
^ 6-25
_prm.h file 1-5
_reg.h file 1-5
| 6-25
|| 6-25
~ 6-23

A
%addincludepath 6-37
array index 6-22
%assert 6-38
assert
  adding B-2
%assign 6-51, 8-23
  defining parameters 4-20

B
block
  customizing Simulink 6-34
block function 8-28
  InitializeConditions 8-33
  Start 8-33
block target file 1-4, 8-28
  function in 8-24
  mapping 4-26
  writing 8-29
BlockInstanceSetup 8-29
block-scoped variable 6-57
BlockTypeSetup 8-30
%body 6-30
Boolean 6-19
%break 6-29, 6-30
%continue 6-29
buffer
  close 6-37
  writing 6-36
built-in functions 6-39
  CAST 6-40
  EXISTS 6-40
  FEVAL 6-40
  FIELDNAMES 6-41
  FILE_EXISTS 6-40

FORMAT 6-41
GENERATE 6-41
GENERATE_FILENAME 6-41
GENERATE_FORMATTED_VALUE 6-41
GENERATE_FUNCTION_EXISTS 6-42
GENERATE_TYPE 6-42
GENERATE_TYPE_FUNCTION_EXISTS 6-42
GET_COMMAND_SWITCH 6-42
GETFIELD 6-41
IDNUM 6-42
IMAG 6-42
INT16MAX 6-42
INT16MIN 6-42
INT32MAX 6-42
INT32MIN 6-42
INT8MAX 6-42
INT8MIN 6-42
INTMAX 6-43
INTMIN 6-43
ISALIAS 6-43
ISEMPTY 6-43
ISEQUAL 6-43
ISFIELD 6-43
ISFINITE 6-43
ISINF 6-43
ISNAN 6-43
NULL_FILE 6-43
NUMTLCFILES 6-43
OUTPUT_LINES 6-44
REAL 6-44
REMOVEFIELD 6-44
ROLL_ITERATIONS 6-44
SETFIELD 6-44
SIZE 6-44
SPRINTF 6-44
STDOUT 6-44
STRING 6-45
STRINGOF 6-45
SYSNAME 6-45
TLC_FALSE 6-45
TLC_TIME 6-46
TLC_TRUE 6-45
TLC_VERSION 6-46
TLCFILES 6-45
TYPE 6-46
UINT16MAX 6-46
UINT32MAX 6-46
UINT8MAX 6-46
UINTMAX 6-46
WHITE_SPACE 6-46
WILL_ROLL 6-46

C
C MEX S-function 1-4
%case 6-29
CAST 6-40
%closefile 6-36
code
  intermediate 4-19
code coverage 7-10
code generation 1-9
coding conventions 8-23
comment
  target language 6-17
CompiledModel 5-3
Compiler
  Target Language (TLC) 1-2
Complex 6-19
Complex32 6-19
conditional
  inclusion 6-28
  operator 6-21

constant
  integer 6-21
  string 6-21
continuation
  line 6-18
%continue 6-30
customizing
  code generation 4-19
  Simulink block 6-34

D

debug
  message 6-38
debugger 7-2, 7-6
  using 7-3
debugger commands
  viewing 7-6
debugging tips 7-2
%default 6-29
Derivatives 8-35
directive 4-20, 6-2
  object-oriented 6-34
  splitting 6-18
directives
  %% 6-2
  %<expr> 6-3
  %addincludepath 6-10
  %addtorecord 6-8
  %assert 6-5
  %assign 6-6
  %break 6-4
  %case 6-4
  %closefile 6-16
  %copyrecord 6-8
  %createrecord 6-7
  %default 6-4
  %else 6-4
  %elseif 6-4
  %endforeach 6-15
  %endfunction 6-13
  %endif 6-4
  %endroll 6-11
  %endswitch 6-4
  %endwith 6-5
  %error 6-6
  %exit 6-6
  %filescope 6-10
  %for 6-16
  %foreach 6-15
  %function 6-13
  %generatefile 6-9
  %if 6-4
  %implements 6-9
  %include 6-10
  %language 6-9
  %matlab 6-2
  %mergerecord 6-8
  %openfile 6-16
  %realformat 6-8
  %return 6-13
  %roll 6-11
  %selectfile 6-16
  %setcommandswitch 6-5
  %switch 6-4
  %trace 6-6
  %warning 6-6
  %with 6-5
  /% text %/ 6-2
Disable 8-31
dynamic scoping 6-55

E
%else 6-28
%elseif 6-28
Enable 8-31
%endbody 6-30
%endfor 6-30
%endforeach 6-29
%endfunction 6-65
%endif 6-28
%endswitch 6-29
%endwith 6-57
%error 6-38
error
  formatting messages B-4
  internal B-2
  usage B-2
error message 6-38
  Target Language Compiler B-5
EXISTS 6-40
%exit 6-39
expressions 6-21
  operators in 6-21
  precedence 6-21

F
FEVAL 6-40
FIELDNAMES 6-41
File 6-19
file
  .c 1-5
  .h 1-5
  .rtw 1-5
  _prm.h 1-5
  _reg.h 1-5
  appending 6-37
  block target 1-4, 4-23
  close 6-37
  inline 6-37
  model description. *See* model.rtw
  model-wide target 4-20
  system target 4-24
  target 1-4, 4-19
  target language 4-26
  used to customize code 4-19
  writing 6-36
FILE_EXISTS 6-40
%for 6-30
%foreach 6-29
FORMAT 6-41
formatting 6-27
Function 6-19
%function 6-65
function
  C MEX S-function 1-4
  call 6-22
  GENERATE 6-35
  GENERATE_TYPE 6-35
  library 8-26
  output 6-66
  target language 6-65
  Target Language Compiler 6-39–6-46
functions
  obsolete 9-2

G
Gaussian 6-19
Gaussian, Unsigned 6-20
GENERATE 6-35, 6-41
GENERATE_FILENAME 6-41
GENERATE_FORMATTED_VALUE 6-41
GENERATE_FUNCTION_EXISTS 6-42
GENERATE_TYPE 6-35, 6-42

GENERATE_TYPE_FUNCTION_EXISTS 6-42
%generatefile 6-34
GET_COMMAND_SWITCH 6-42
GETFIELD 6-41

I
identifier 8-23
  changing 6-51
  defining 6-51
IDNUM 6-42
%if %endif 6-28
IMAG 6-42
%implements 6-34
%include 6-37
inclusion
  conditional 6-28
  multiple 6-29
index 6-22
Initialize 8-33
InitializeConditions 8-33
inlining S-function 8-4
  advantages 1-13
input file control 6-37
INT16MAX 6-42
INT16MIN 6-42
INT32MAX 6-42
INT32MIN 6-42
INT8MAX 6-42
INT8MIN 6-42
integer constant 6-21
intermediate code 4-19
INTMAX 6-43
INTMIN 6-43
ISALIAS 6-43
ISEMPTY 6-43
ISEQUAL 6-43
ISFIELD 6-43
ISFINITE 6-43
ISINF 6-43
ISNAN 6-43

L
%language 6-34
lcv, definition 9-4
library functions
  LibBlockContinuousState 9-28
  LibBlockDiscreteState 9-29
  LibBlockDWork 9-28
  LibBlockDWorkAddr 9-28
  LibBlockDWorkDataTypeId 9-28
  LibBlockDWorkDataTypeName 9-28
  LibBlockDWorkIsComplex 9-29
  LibBlockDWorkName 9-29
  LibBlockDWorkStorageClass 9-29
  LibBlockDWorkUsedAsDiscreteState 9-29
  LibBlockDWorkWidth 9-29
  LibBlockInputSignal 9-9
  LibBlockInputSignalAddr 9-16
  LibBlockInputSignalBufferDstPort 9-47
  LibBlockInputSignalConnected 9-16
  LibBlockInputSignalDataTypeId 9-17
  LibBlockInputSignalDataTypeName 9-17
  LibBlockInputSignalDimensions 9-17
  LibBlockInputSignalIsComplex 9-17
  LibBlockInputSignalIsFrameData 9-17
  LibBlockInputSignalLocalSampleTimeIndex
    9-18
  LibBlockInputSignalNumDimensions 9-18
  LibBlockInputSignalOffsetTime 9-18
  LibBlockInputSignalSampleTime 9-18
  LibBlockInputSignalSampleTimeIndex 9-18
  LibBlockInputSignalStorageClass 9-48

LibBlockInputSignalStorageTypeQualifier 9-48
LibBlockInputSignalWidth 9-18
LibBlockIWork 9-29
LibBlockMatrixParameter 9-23
LibBlockMatrixParameterAddr 9-23
LibBlockMode 9-30
LibBlockNonSampledZC 9-30
LibBlockOutputSignal 9-19
LibBlockOutputSignalAddr 9-19
LibBlockOutputSignalBeingmerged 9-20
LibBlockOutputSignalConnected 9-20
LibBlockOutputSignalDataTypeId 9-20
LibBlockOutputSignalDataTypeName 9-20
LibBlockOutputSignalDimensions 9-21
LibBlockOutputSignalIsComplex 9-21
LibBlockOutputSignalIsFrameData 9-21
LibBlockOutputSignalIsGlobal 9-48
LibBlockOutputSignalIsInBlockIO 9-49
LibBlockOutputSignalIsValidLValue 9-49
LibBlockOutputSignalLocalSampleTimeIndex 9-21
LibBlockOutputSignalNumDimensions 9-21
LibBlockOutputSignalOffsetTime 9-21
LibBlockOutputSignalSampleTime 9-22
LibBlockOutputSignalSampleTimeIndex 9-22
LibBlockOutputSignalStorageClass 9-49
LibBlockOutputSignalStorageTypeQualifier 9-49
LibBlockOutputSignalWidth 9-22
LibBlockParameter 9-24
LibBlockParameterAddr 9-25
LibBlockParameterBaseAddr 9-26
LibBlockParameterDataTypeId 9-26
LibBlockParameterDataTypeName 9-26
LibBlockParameterDimensions 9-26
LibBlockParameterIsComplex 9-27
LibBlockParameterSize 9-27
LibBlockPWork 9-30
LibBlockReportError 9-31
LibBlockReportFatalError 9-31
LibBlockReportWarning 9-31
LibBlockRWork 9-30
LibBlockSampleTime 9-37
LibBlockSrcSignalBlock 9-49
LibBlockSrcSignalIsDiscrete 9-50
LibBlockSrcSignalIsGlobalAndModifiable 9-50
LibBlockSrcSignalIsInvariant 9-51
LibCacheDefine 9-33
LibCacheExtern 9-34
LibCacheFunctionPrototype 9-34
LibCacheIncludes 9-35
LibCacheTypedefs 9-35
LibCallFCSS 9-43
LibGetBlockPath 9-32
LibGetDataTypeAliasedToFromId 9-44
LibGetDataTypeComplexNameFromId 9-44
LibGetDataTypeEnumFromId 9-44
LibGetDataTypeNameFromId 9-45
LibGetDataTypeResolvesToFromId 9-44
LibGetFormattedBlockPath 9-32
LibGetGlobalTIDFromLocalSFcnTID 9-37
LibGetNumSFcnSampleTimes 9-39
LibGetSFcnTIDType 9-39
LibGetT 9-45
LibGetTaskTimeFromTID 9-39
LibIsComplex 9-45
LibIsContinuous 9-39
LibIsDiscrete 9-40
LibIsFirstInitCond 9-46
LibIsSFcnSampleHit 9-40
LibIsSFcnSingleRate 9-41

  LibIsSFcnSpecialSampleHit 9-41
  LibMaxIntValue 9-46
  LibMinIntValue 9-46
  LibSetVarNextHitTime 9-42

M
macro
  expansion 6-22
makefile
  template 1-4
Matrix 6-19
mdlDerivatives (S-function) 8-4
mdlInitializeConditions 8-4
mdlInitializeSampleTimes 8-4
mdlInitializeSizes 8-4
mdlOutputs (S-function) 8-4
MdlStart
  InitializeConditions 8-33
MdlTerminate
  Terminate 8-35
mdlTerminate (S-function) 8-4
mdlUpdate (S-function) 8-4
model description file. *See* model.rtw
model.rtw file
  changing scope within A-5
model.rtw file 1-3, 6-39
model-wide target file 4-20
modifier
  Output 6-66
  void 6-66
multiple inclusion 6-29

N
negation operator 6-23
nested function
  scope within 6-67
new features
  version 4.0 1-18
  version 4.1 1-17
  version 5.0 1-16
NULL_FILE 6-43
Number 6-19
NUMTLCFILES 6-43

O
object-oriented directive 6-34
obsolete functions 9-2
%openfile 6-36
operations
  precedence 6-23
operator 6-21
  :: 8-24
  conditional 6-21
  negation 6-23
output file control 6-36
Output modifier 6-66
OUTPUT_LINES 6-44
Outputs 8-33

P
parameter
  defining 4-20
  value pair 5-2
paramIdx 9-6
path
  specifying absolute 6-38
  specifying relative 6-38
portIdx, definition 9-4

precedence
  expressions 6-21
  operations 6-23
profiler 7-13
  using 7-13
program 4-26

R
Range 6-20
REAL 6-44
Real 6-20
Real32 6-20
%realformat 6-27
Real-Time Workshop 1-2
record 4-14, 5-2
record file
  example of 3-4
  guide.rtw 3-4
  lists 3-11
  reading 3-3
  structure of 3-3
REMOVEFIELD 6-44
resolving variables 6-55
%return 6-65, 6-69
%roll
  common arguments to 9-4
  directive 6-11
  syntax of 6-31
ROLL_ITERATIONS 6-44
rt 8-25
rt_ 8-25
RTW
  identifier 8-23

S
Scope 6-20
scope 6-55
  accessing values in 5-3
  closing 6-69
  dynamic 6-55
  function 6-22
  modifying via %width 3-15
  within function 6-67
search path 6-72
  adding to 6-38
  overriding 6-72
  sequence 6-38
  specifying absolute 6-38
  specifying relative 6-38
%selectfile 6-36
SETFIELD 6-44
S-function
  adavantage of inlining 1-13
  C MEX 1-4
  inlining 8-4
  user-defined 8-35
S-function record A-105
sigIdx 9-5
Simulink
  and Real-Time Workshop 1-2
  generating code 1-5
Simulink data objects
  and ObjectProperties records 5-6
SIZE 6-44
Special 6-20
SPRINTF 6-44
Start 8-32
stateIdx 9-6
STDOUT 6-44
STRING 6-45
String 6-20

string constant 6-21
STRINGOF 6-45
substitution
  textual 6-21
Subsystem 6-20
%switch 6-29
syntax 6-2
SYS_NAME 6-45
system target file 4-24

T
target file 1-4, 4-19, 4-23
  and customizing code 4-19
  block 4-25, 8-28
  model-wide 4-20
  naming 6-72
  system 4-24
target language 4-13
  comment 6-17
  directive 4-20, 6-2
  expression 6-21–6-26
  file 6-2
  formatting 6-27
  function 6-65
  line continuation 6-18
  program 4-26
  syntax 6-2
  value 6-19–6-20
Target Language Compiler
  command line arguments 6-70
  directives 6-2–6-17
  error messages B-5
  function library 8-26
  introducing 1-2
  switches 6-70
  uses of 1-7
  variables 8-25
template makefile 1-4
Terminate 8-35
textual substitution 6-21
TLC code
  debugging tips 7-2
TLC coverage option 7-10
TLC debugger 7-2
TLC debugger commands 7-6
TLC profiler 7-13
TLC program 4-26
TLC tutorials
  debugging TLC code 3-40
  inlining S-functions 3-25
  processing rtw files 3-18
  reading record files 3-3
  using TLC code coverage data 3-49
  variable names and loop rolling 3-32
  wrapping user code 3-53
TLC_FALSE 6-45
TLC_TIME 6-46
TLC_TRUE 6-45
TLC_VERSION 6-46
TLCFILES 6-45
token expansion 3-7
%trace 6-39
tracing 6-39
TYPE 6-46

U

ucv, definition 9-4
UINT16MAX 6-46
UINT32MAX 6-46
UINT8MAX 6-46
UINTMAX 6-46
Unsigned 6-20
Unsigned Gaussian 6-20
Update 8-35

V

values 6-19
variables
    block-scoped 6-57
    global 8-24
    local 8-24
Vector 6-20
void modifier 6-66

W

%warning 6-38
warning message 6-38
WHITE_SPACE 6-46
WILL_ROLL 6-46
%with 6-57

Z

zero-crossing
    reset code 8-35