SIMULINK®

Dynamic System Simulation for MATLAB®

Modeling

Simulation

Implementation

Target Language Compiler Reference Guide

Version 1

How to Contact The MathWorks:

508-647-7000	Phone

508-647-7001	Fax

The MathWorks, Inc. 24 Prime Park Way Natick, MA 01760-1500	Mail

`http://www.mathworks.com`	Web
`ftp.mathworks.com`	Anonymous FTP server
`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
`subscribe@mathworks.com`	Subscribing user registration
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

Target Language Compiler Reference Guide

Printing History: May 1997 First printing for Target Language Compiler 1.0

Contents

Using the Target Language Compiler with Real-Time Workshop

1

Working with the Target Language

2

3 Writing Target Language Files

Target Language Compiler Function Library Reference

4

model.rtw

A

Target Language Compiler Error Messages

B

Target Language Compiler Library Error Messages

C

1

Using the Target Language Compiler with Real-Time Workshop

Using the Target Language Compiler

Introduction

The Target Language Compiler™ is a tool that is included with Real-Time Workshop® (RTW) and enables you to customize the C code generated from any Simulink® model. Through customization, you can produce platform-specific code or incorporate algorithmic changes for performance, code size, or compatibility with existing methods that you prefer to maintain.

Notes: This book describes the Target Language Compiler, its files, and how to use them together. This information is provided for those users who need to customize target files in order to generate specialized output. Or, in some cases, for users who want to inline S-functions so as to improve the performance of the generated code. If you simply need to generate ANSI C code from your Simulink model, you can find everything you need to know in the *Real-Time Workshop User's Guide*.

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

As an integral component of Real-Time Workshop, the Target Language Compiler is used to transform an intermediate form of a Simulink block diagram, called *model*`.rtw`, into C code. The 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 Compiler 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. A target-specific version of the generic `rt_main` file (or `grt_main`) must also be modified to conform to the target's specific requirements such as interrupt service routines. A complete description of the template makefiles and `rt_main` is included in the *Real-Time Workshop User's Guide*.

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

The code generated by the Compiler is highly optimized and fully commented C code, and can be generated from any Simulink model, including linear, nonlinear, continuous, discrete, or hybrid. All Simulink blocks are automatically converted to code, with the exception of MATLAB function blocks and S-function blocks that invoke M-files. The Target Language Compiler uses "block target files" to transform each block in the *model*.rtw file and a "model-wide target file" for global customization of the code.

You can incorporate C MEX S-functions, along with the generated code, into the program executable. You can also write a target file for your C MEX S-function to "inline" the S-function, thus improving performance by eliminating function calls to the S-function itself. Inlining an S-function incorporates the S-function block's code into the generated code for the model. When no target file is present for the S-function, its C code file is invoked via a function call. For more information on inlining S-functions, see "Inlining an S-Function," in Chapter 3. You can also write target files for M-files or Fortran S-functions.

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 the *model*.rtw file in combination with a set of included target files to generate the body source C code (*model*.c), a header file (*model*.h), a model registration include file (*model*.reg) that registers the model's SimStruct, and a parameter include file (*model*.prm) that contains information about all the parameters contained in the model.

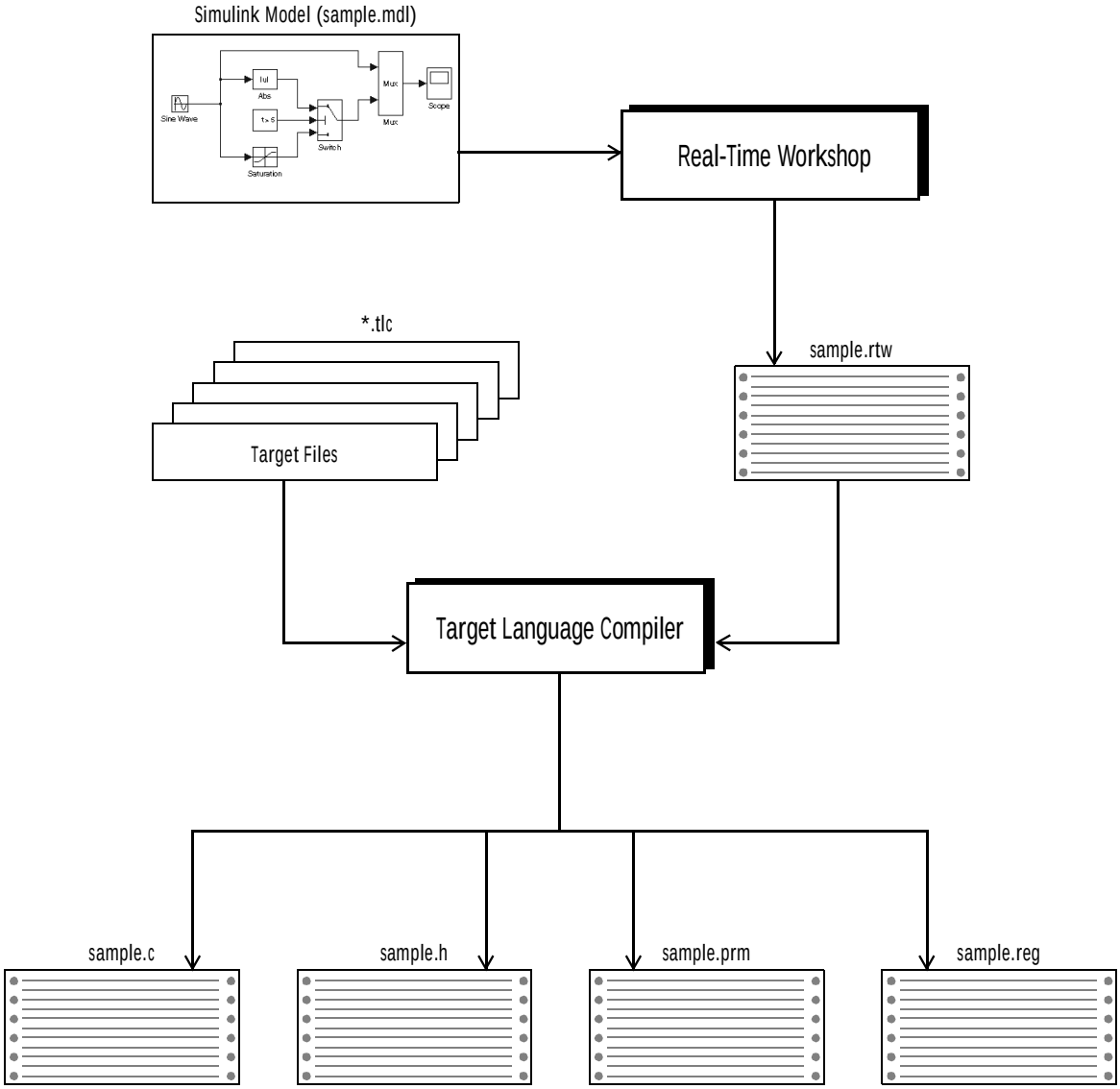


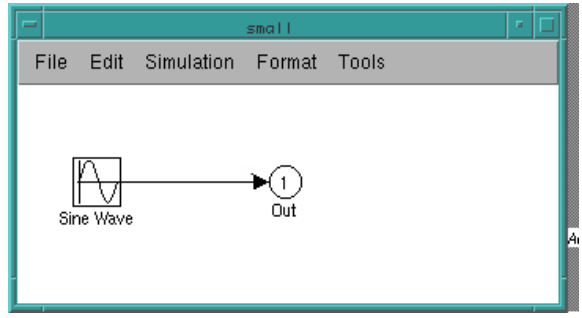
Figure 1-1: The Target Language Compiler Process

By modifying the target files (also referred to as TLC files, for example, `basic.tlc`), you can customize the output generated by the Target Language Compiler. For more information on creating TLC files, see Chapter 3, "Writing Target Language Files."

A Basic Example

The Real-Time Workshop graphical user interface provides a menu that automates the entire process of generating a *model*`.rtw` file, invoking the Target Language Compiler, and running the build process to generate an executable. This example uses command line operation of the Target Language Compiler in the same way that RTW's graphical user interface invokes the Target Language Compiler to show the step-by-step process.

Starting with a simple Simulink model named `small.mdl`, this example illustrates how to use the Target Language Compiler on the output that is generated from the model.



Using this model, Real-Time Workshop generates an `.rtw` file, `small.rtw`, containing all of the model's information. To generate the file from the command line, enter:

```
rtwgen small
```

This file shows the structure of the `small.rtw` file.

```
CompiledModel {
  Name                          "small"
  Version                       "2.01 (April, 1 1997)"
  GeneratedOn                   "Fri Apr 11 11:31:25 1997"
  Solver                        FixedStepDiscrete
  SolverType                    FixedStep
  StartTime                     0
  StopTime                      10
       <Parameters omitted>
BlockOutputs {
    NumBlockOutputs             1
    BlockOutput {
      Identifier                Sine_Wave
      SigSrc                    [0, 0, 0]
      SigIdx                    [0, 1]
      SigConnected              [1]
      SigLabel                  ""
      TestPoint                 no
      Required                  no
    }
  }
  BlockOutputsMap               Matrix(1,2)
[[0, 0];]
       <BlockDefaults omitted>
System {
    Type                        root
    Name                        "<root>"
    Identifier                  root
    NumBlocks                   2
    Block {
      Type                      Sin
      Name                      "<Root>/Sine Wave"
      Identifier                Sine_Wave
      TID                       0
      RollRegions               [0]
      NumDataOutputPorts        1
      DataOutputPortIndices     [0]
      Parameters                [3, 3, 0]
```

```
        <Parameters omitted>
}
    Block {
      Type                  Outport
      Name                  "<Root>/Out"
      Identifier            Out
      TID                   0
      RollRegions           [0]
      NumDataInputPorts     1
      DataInputPort {
    Width                   1
    SignalSrc               [B0]
      }
      ParamSettings {
    PortNumber              1
    OutputLocation          Y0
      }
    }
  }
    .
    .
    .
}
```

To use the Target Language Compiler on `small.rtw` to generate all associated RTW code, enter:

```
tlc –r small.rtw MATLAB/rtw/c/grt/grt.tlc –IMATLAB/rtw/c/tlc
```

Note: To use the Target Language Compiler and its associated files, you must know where MATLAB is installed on your system. MATLAB provides a command that returns this information. Whenever you see the directory *MATLAB* in this manual, you should replace it with the path returned by the `matlabroot` command. For example, if `matlabroot` returns:

```
matlabroot
ans =
/usr/apps/matlab
```

you would use the command:

```
tlc –r small.rtw /usr/apps/matlab/rtw/c/grt/grt.tlc
–I/usr/apps/matlab/rtw/c/tlc
```

The Target Language Compiler processes `small.rtw` using the system target file, `grt.tlc`, along with other system target files to generate the RTW code. The generated output consists of the files: `small.h`, `small.prm`, `small.c`, and `small.reg`.

File	Purpose
`small.c`	Source file implementing the algorithms defined by your model.
`small.h`	Header file containing structure declarations. This file is included by `small.c`, `small.prm`, and `small.reg`.
`small.prm`	Include file containing the default parameters and global data declarations. This file is included once at the top of `small.c`.
`small.reg`	Include file containing the model registration function and other initialization routines. This file is included once at the bottom of `small.c`.

These files contain the fully documented C code that represents your Simulink model. At this point, you can use the C code as a stand-alone external simulation on a target machine.

This example shows only the basic operation of the Target Language Compiler. Numerous options are available and are explained throughout this manual.

Files

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

Target Files

Target files are the set of files that are interpreted by the Target Language Compiler to transform the intermediate RTW 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 `mdlwide.tlc`, you produce generic C code from your Simulink model. This executable C code is not platform specific. By modifying `mdlwide.tlc`, or creating a completely new target file, you could output C code specific to a particular piece of hardware, or for that matter, output code in another language.

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 Chapter 2.

Target files are written using target language directives. Chapter 2, “Working with the Target Language,” provides complete descriptions of the target language directives.

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

System Target Files

System target files are used on a model-wide basis and provide basic information to the Target Language Compiler, which transforms the RTW file into target-specific code. The system target file is the “entry point” for the TLC program, which is analogous to the `main()` routine of a C program. System target files oversee the entire code generation process. For example, the system target file, `grt.tlc`, sets up some variables for `mdlwide.tlc` and includes

`mdlwide.tlc`, which contains all of the settings and parameter values that control generic C code generation.

The set of system target files includes:

System Target File	Purpose
`grt.tlc`	System target file for generic real-time code generator
`mdlwide.tlc`	General C code target file provided by The MathWorks
`mdlhdr.tlc`	Creates everything in the header file, *model*`.h`
`mdlbody.tlc`	Creates the source file, *model*`.c`
`mdlreg.tlc`	Creates the model registration file, *model*`.reg`
`mdlparam.tlc`	Creates the parameters file, *model*`.prm`

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.

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

Where to Go from Here

The remainder of this book contains both explanatory and reference material for the Target Language Compiler. Use this chart to help determine which chapters are most relevant for you.

Chapter	Description
1. Using the Target Language Compiler with Real-Time Workshop	Provides overview information of the Target Language Compiler and its files.
2. Working with the Target Language	Provides a complete description of the constructs used to create target language files and general coding guidelines.
3. Writing Target Language Files	Describes the process of customizing target files and inlining S-functions.
4. Target Language Compiler Function Library Reference	Provides complete descriptions of all functions used to create block target files.
Appendix A. model.rtw	Complete description of the *model*.rtw file generated by Real-Time Workshop build procedure.
Appendix B. Target Language Compiler Error Messages	Error messages generated by the Target Language Compiler and their descriptions.
Appendix C. Target Language Compiler Library Error Messages	Error messages generated when working with the Target Language Compiler libraries and their descriptions.

2

Working with the Target Language

Why Use the Target Language Compiler?

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

- You need to change the way code is generated for a particular Simulink block.
- You need to inline S-functions in your model.
- You need to modify the way code is generated in a global sense.
- You need to perform a large scale customization of the generated code. For example, you need to output the code in a language other than C.

To produce customized output using the Target Language Compiler, you need to understand the structure of the *model*`.rtw` file and how to modify target files to produce the desired output. This chapter first introduces the *model*`.rtw` file and then describes the target language directives and their associated constructs. You will use the TLC directives and constructs to modify existing target files or create new ones from scratch, depending on your needs. Chapter 3 explains the details of writing target files.

The model.rtw File

Real-Time Workshop generates a *model*.rtw file from your Simulink model. The *model*.rtw file is a hierarchical database whose contents provide a description of the individual blocks within the Simulink model.

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 RTW identifier and value is a string, scalar, vector, or matrix. For example, in the parameter-value pair

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

NumDataOutputPorts is the identifier and 1 is its value.

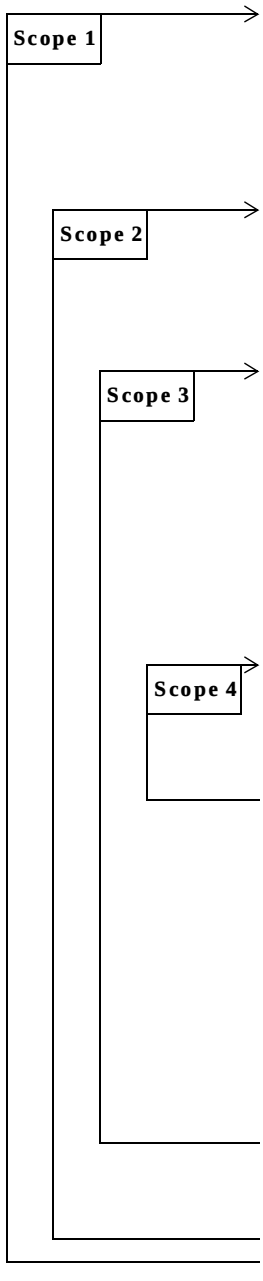
A record is specified as:

```
RecordName {
      .
      .
}
```

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

```
DataStores {
    NumDataStores         0
}
```

The following reduced example shows a record, Block, with 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.




```
CompiledModel {
  Name                      "gain1"
  Version                   "1.85 (Oct, 30 1996)"
    .
    .
    .
  System {
    Type                    root
    Name                    "<root>"
    Identifier              root
    NumBlocks               3
    Block {
      Type                  Sin
      Name                  "<Root>/Sine Wave"
      Identifier            Sine_Wave
      TID                   0
      RollRegions           [0]
      NumDataOutputPorts    1
      DataOutputPortIndices [0]
      Parameters            [3, 3, 0]
      Parameter {
        Name                "Amplitude"
        Value               [1]
        String              "1"
      }
      Parameter {
        Name                "Frequency"
        Value               [1]
        String              "1"
      }
      Parameter {
        Name                "Phase"
        Value               [0]
        String              "0"
      }
    }
  .
  .
  }
}
```

This example shows several records and corresponding subrecords by use of arrows. `Parameter` (Scope 4) is a subrecord of `Block` (Scope 3), which is a subrecord of `System` (Scope 2), which in turn is a subrecord of `CompiledModel` (Scope 1).

The *model*`.rtw` file uses curly braces `{` and `}` to open and close scopes. Using scopes, you can access any value within the *model*`.rtw` file. The scope in this example begins with `CompiledModel`. You use periods (`.`) to access values within particular scopes. For example, to access `Name` within `CompiledModel`, you would use

```
CompiledModel.Name
```

To access `Identifier` within `System` within `CompiledModel`, you would use

```
CompiledModel.System.Identifier
```

To access `Name` within the second `Parameter` record within `Block` within `System` within `CompiledModel`, you would use

```
CompiledModel.System.Block.Parameter[2].Name
```

This process can be simplified by using the `%with` directive. See the “Scoping” section later in this chapter for more information.

The identifier and record name become TLC variables when the Target Language Compiler loads the *model*`.rtw` file.

The Target Language Compiler lets you traverse the hierarchy defined by *model*`.rtw` so that you can customize the output to suit your particular needs. To be able to do this, you must understand the structure of the *model*`.rtw` file. Appendix A contains a complete description of the *model*`.rtw` file.

Compiler Directives

Syntax

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

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

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

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

uses the `%assign` directive to change the value of the `sysNumber` parameter. A target language directive must be the first nonblank character on a line and always begins with the `%` character. Beginning a line with `%%` lets you include a comment on a line.

Table 2-1 shows the complete set of TLC directives. The remainder of this chapter describes each directive in detail.

Table 2-1: Target Language Compiler Directives

Type	Construct
Comments	`/% %/` `%%`
Target language expressions	`%<`*`expression`*`>`
Formatting	`%realformat` *`string`*

Table 2-1: Target Language Compiler Directives (Continued)

Type	Construct

Conditional inclusion	%if *constant-expression*
	%else
	%elseif *constant-expression*
	%endif
	
	%switch *constant-expression*
	  %case *constant-expression*
	  %default
	  %break
	%endswitch

Multiple inclusion	%foreach *identifier = constant-expression*
	  /* Loops from 0 to N – 1 */
	  %break
	  %continue
	%endforeach
	
	%roll *identifier = roll-vector-expression, identifier = threshold-expression, block-expression* [, *type-string* [,*expression-list*]]
	  %break
	  %continue
	%endroll
	
	%for identifier = *constant-exp, constant-exp, identifier*
	%body
	  %break
	  %continue
	%endbody
	%endfor

Object-oriented facility	%generatefile *identifier string* %language %implements

Table 2-1: Target Language Compiler Directives (Continued)

Type	Construct
Output file control	`%openfile` *x optional-string "optional-mode"* `%closefile` `%selectfile` *identifier*
Input file control	`%include` *string* `%addincludepath` *string*
Debug statements	`%error` *tokens* `%warning` *tokens* `%trace` *tokens* `%exit` *tokens*
Macro definition	`%define` *identifier opt-argument-list replacement-list* `%undef`
Identifier definition	`%assign [::]`*expression = constant-expression*
Scoping	`%with` *expression* `%endwith`

Target language functions	`%function` *identifier (optional-arguments) [Output | void]*
	`%endfunction`
	`%return`

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 the "Target Language Functions" section in this chapter.

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
```

Target Language Values

Table 2-2 shows the types of values you can use within the context of expressions in your target language files:

Table 2-2: Target Language Values

Value Type String	Example	Description
"Boolean"	1==1	Result of a comparison or other Boolean operator. **Note:** There are no Boolean constants, and Boolean values are 1 or 0 as in C. 1 is still a number and not a Boolean value.
"File"	%openfile x	String buffer opened with %openfile.
"File"	%openfile x = "out.c"	File opened with %openfile.
"Function"	%function foo…	A user-defined function.
"Identifier"	abc	Identifier values can only appear within the .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.
"Macro"	%define MACRO …	A user-defined macro.

Table 2-2: Target Language Values (Continued)

Value Type String	Example	Description
"Matrix"	Matrix (3,2) [[1, 2] [3 , 4] [5, 6]]	Matrices are simply lists of vectors. The individual elements of the matrix do not need to be the same type, and can be any type except vectors or matrices.
"Number"	15	An integer number.
"Range"	1:5	A range of integers between 1 and 5, inclusive, cannot be specified except in the .rtw file or vector because of syntactic ambiguity with the ? : operator. Use [1:5][0] to generate a range.
"Real"	3.14159	A floating-point number (including exponential notation).
"Scope"	Block { … }	A block-scope.
"Special"	N/A	A special built-in function, such as FILE_EXISTS.

Table 2-2: Target Language Values (Continued)

Value Type String	Example	Description
"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.
"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\>>.
"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 Compiler replaces expression with a calculated replacement value based upon its type. 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 a parameter or function argument currently in scope.

You can use the %< > directive on any line to perform textual substitution. To include the > character within a replacement, you must escape it with a "\" character as in:

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

Note: It is not necessary to place expressions in the %< > format when they appear on directive lines.

Table 2-3 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 opposed to C expressions, conditional operators are not short-circuited. Therefore, if the expression includes a function call with side effects, the effects are noticed as if the entire expression was evaluated.

In the Target Language Compiler, you cannot depend on short-circuit evaluation to avoid errors such as:

```
%if EXISTS(foo) && foo == 3
```

This statement would cause an error if `foo` was undefined.

Table 2-3: 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. See “Identifier Definition” on page 2-37.
`expr[expr]`	Index into an array parameter. Array indices range from `0` to `N–1`. This syntax is used to index into vectors, matrices, and repeated scope variables.
`expr([expr[,expr]...])`	Function call or macro expansion. The expression outside of the parentheses is the function/macro name; the expressions inside are the arguments to the function or macro. **Note:** Since macros are text-based, they cannot be used within the same expression as other operators.
`expr. expr`	The first expression must be 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 `1` or `0` as in C). The argument must be numeric or Boolean.

Table 2-3: Target Language Expressions (Continued)

Expression	Definition
–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.
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 adds 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 adds 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).

Table 2-3: Target Language Expressions (Continued)

Expression	Definition
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.
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.

Table 2-3: Target Language Expressions (Continued)

Expression	Definition
`expr && expr`	Performs the logical AND of the two arguments and returns `1` or `0`. This can be used on either numeric or Boolean arguments.
`expr || expr`	Performs the logical OR of the two arguments and returns `1` or `0`. This can be used on either numeric or Boolean arguments.
`expr ? expr : expr`	Tests the first expression for logical truth. If true, the first expression is returned; otherwise the second expression is returned. **Note:** Both are evaluated.
`expr , expr`	Returns the value of the second expression.

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 the “Target Language Expressions” section.

%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

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 only includes 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, *identifier1* 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.

`identifier2` 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.

Note: The `%for` directive is functional, but it is not recommended. Rather, use `%roll`, which provides the same capability in a more open way. RTW does not make use of the `%for` construct.

%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 RTW has the capability to extract references to the Block I/O and other RTW-specific vectors that vastly simplify the body of the `%roll` statement. These functions include `LibBlockInputSignal`, `LibBlockOutputSignal`, `LibBlockParameter`, `LibBlockRWork`, `LibBlockIWork`, `LibBlockPWork`, and `LibDeclareRollVars`. For more details on these functions and other Simulink functions, see the section on Loop Rolling beginning on page 3-44 along with the "Target Language Compiler Function Library Reference" chapter. This library also includes a default implementation of `Roller.tlc`.

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 TLC code produces the output:

```
{
    int             i;
    for (i = 1; i < 21; i++)
    {
        Block[i] *= 3.0;
    }
    Block[21] *= 3.0;
    Block[22] *= 3.0;
    Block[23] *= 3.0;
    Block[24] *= 3.0;
    Block[25] *= 3.0;
    for (i = 26; i < 47; i++)
    {
        Block[i] *= 3.0;
    }
}
```

Object-Oriented Facility for Generating Target Code

The Target Language Compiler provides a simple object-oriented facility. The language directives are:

```
%language string
%generatefile
%implements
```

This facility was designed specifically for customizing the code for Simulink blocks, but can be used for other purposes as well.

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" ["Ada", "Pascal"]
```

causes an error if the initial language choice was C.

You can use a single file to implement more than one target language by specifying the desired languages in a vector. For example:

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

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

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

Note: The use of the wildcard (*) is not recommended because it relaxes error checking for the %implements directive.

GENERATE and GENERATE_TYPE Functions

The Target Language Compiler has two built-in functions that dispatch object-oriented calls, GENERATE and GENERATE_TYPE. You can call any function

appearing in an implementation file (from outside the specified file) only by using the GENERATE and GENERATE_TYPE special functions.

The GENERATE function takes two or more input arguments. The first argument must be a valid scope and the second a string containing the name of the function to call. The GENERATE function passes the first block argument and any additional arguments specified to the function being called. The return argument is the value (if any) returned from the function being called. Note that the Compiler automatically "scopes" or adds the first argument to the list of scopes searched as if it appears on a %with directive line. See %with in "Scoping" beginning on page 2-41. This scope is removed when the function returns.

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 directive 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
```

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. %/
```

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. %/
```

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
```

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.

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)
%addincludepath "directory1:directory2"       (Macintosh)
```

To specify a relative path, the path must explicitly start with "." on the PC or UNIX, or ":" on the Macintosh. For example:

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

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.

Errors, Warnings, and Debug Messages

The related error, warning, and debug message directives are:

```
%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 Compiler places messages generated by `%trace` onto `stderr` if and only if you specify the verbose mode switch (`-v[1|2|3]`) to the Compiler. See the section "Command Line Arguments," later in this chapter for additional information about switches.

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

Built-In Functions and Values

Table 2-4 lists the built-in functions and values that are added to the list of parameters that appear in the `.rtw` file. These TLC functions and values are

defined in uppercase so that they are visually distinct from other parameters in the `.rtw` file, and by convention, from user-defined parameters.

Table 2-4: TLC Built-in Functions and Values

Special Macro 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. One application of this is to allow outputs to be generated as floating-point values.
`EXISTS(expr)`	`expr` must be a string. If the identifier is not currently in scope, the result is `0`. If the identifier is in scope, the result is `1`. `expr` can be a single identifier or an expression involving the `.` and `[]` operators.
`FEVAL(expr1, expr2)`	Performs an evaluation in MATLAB. See the "FEVAL Function" section starting on page 2-35 for more information.
`FILE_EXISTS(expr)`	`expr` must be a string. If a file by the name `expr` does not exist on the path, the result is `0`. If a file by that name exists on the path, the result is `1`.

Table 2-4: TLC Built-in Functions and Values (Continued)

Special Macro Name	Expansion
`FORMAT(expr1, expr2)`	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.
`GENERATE(expr1, expr2, ...)`	See the description in the "Object-Oriented Facility for Generating Target Code" section, on page 2-24.
`GENERATE_FILENAME(expr)`	Treats the expression as a `Type`, and returns the name of the `.tlc` file that will be opened for that `Type`.
`GENERATE_FUNCTION_EXISTS (expr, expr)`	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 (expr1, expr2, expr3)`	See the description in the "Object-Oriented Facility for Generating Target Code" section, on page 2-24.

Table 2-4: TLC Built-in Functions and Values (Continued)

Special Macro Name	Expansion
`GENERATE_TYPE_FUNCTION_EXISTS (expr1, expr2, expr3)`	Same as `GENERATE_FUNCTION_EXISTS` except it overrides the `Type` built into the object. See the description of `GENERATE_TYPE` for more information.
`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]`
`NULL_FILE`	A predefined file for no output that you can use as an argument to `%selectfile` to prevent output.
`NUMTLCFILES`	The number of target files used thus far in expansion.
`OUTPUT_LINES(expr)`	Accepts a file variable as input and returns the number of lines that have been written to the given file or buffer.

Table 2-4: TLC Built-in Functions and Values (Continued)

Special Macro Name	Expansion
`SIZE(expr[,expr])`	Calculates the size of the first expression and generates a two-element, row vector. If the second operand is specified, it is used as an integral index into this row vector; otherwise the entire row vector is returned. `SIZE(x)` applied to any scalar returns `[1 1]`. `SIZE(x)` applied to any scope returns the number of repeated entries of that scope type (e.g., `SIZE(Block)` returns `[1,<number of blocks>]`.
`STAND_ALONE`	This can be used to determine if the `FEVAL` function is available. When running from MATLAB, its value is 0; when running from the shell, its value is 1.
`STDOUT`	A predefined file for `stdout` output. You can use as an argument to `%selectfile` to force output to stdout.
`STRING(expr)`	Expands the expression into a string; the characters \, \n, and " are escaped by preceding them with \ (backslash). All the ANSI escape sequences are translated into string form.

Table 2-4: TLC Built-in Functions and Values (Continued)

Special Macro Name	Expansion
`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 RTW.

`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 RTW. For example:
	`%<sysname("<sub>/Gain")>`
	returns
	`["sub","Gain"]`
	To expand a full Simulink path name, see `LibPathName` in the "Target Language Compiler Function Library Reference" chapter.

`TLCFILES`	Returns a vector containing the names of all the target files included thus far in the expansion. Also, see `NUMTLCFILES`.
`TLC_TIME`	The date and time of compilation.
`TLC_VERSION`	The version and date of the Target Language Compiler.

Table 2-4: TLC Built-in Functions and Values (Continued)

Special Macro Name	Expansion
`TYPE(expr)`	Evaluates `expr` and determines the result type. The result of this function is a string that corresponds to the type of the given expression. See value type string in the Target Language Values table for possible values.
`WHITE_SPACE(expr)`	Accepts a string and returns 1 if the string contains only whitespace characters (, \t, \h, \r) and 0 otherwise.
`WILL_ROLL(expr1, expr2)`	The first expression is a roll vector and the second expression is a threshold. This function returns `true` if the vector contains a range that will roll.

FEVAL Function

The `FEVAL` built-in function calls MATLAB M-file functions and MEX-functions. The structure is:

```
%assign result = FEVAL( matlab-function-name, rhs1, rhs2, ...
                        rhs3, ... );
```

Note: Only a single left-hand-side argument is allowed when calling MATLAB.

When calling MATLAB, these conversions are made:

TLC Type	MATLAB Type
"Boolean" or "Number" or "Real"	Double Scalar
"String"	Char Vector
"Vector"	If the vector is entirely strings, then Char Matrix. If it is entirely numeric, then Double Vector. Otherwise, it is an error.

When values are returned from MATLAB, they are converted as follows:

MATLAB Type	TLC Type
Double Scalar	"Number" or "Real" depending on the value
Char Row Vector	"String"
Char Matrix or Char Col Vector	"Vector" of "Strings"
Double Vector	"Vector" whose elements are "Number" or "Real" depending on their values

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 )
```

Note: The FEVAL function is only available from the MATLAB command line version of the Target Language Compiler. It is not available from the shell version. Use the STAND_ALONE predefined value to determine if FEVAL is available to you.

Macro Definition

To simplify complicated references, target files can define macros that are expanded when they appear in subsequent expressions.

```
%define identifier opt-argument-list replacement-list
```

To undefine a previously defined macro, use:

```
%undef identifier
```

identifier is the name of the macro being defined or undefined; *opt-argument-list* is either a C macro argument list or is omitted; *replacement-list* is an expansion list similar to a C language macro.

Note: This facility works, but it is not recommended. Rather, use `%assign` and `%function`, which provide the same capabilities in a more open way. RTW does not make use of macros.

Identifier Definition

To define or change identifiers (TLC variables), use the directive:

```
%assign [::]expression = constant-expression
```

This directive introduces new identifiers (variables) or changes the values of existing ones. The left-hand side can be a qualified reference to a variable using the `.` and `[]` operators, or it can be a single element of a vector or matrix. In the case of the matrix, only the single element is changed by the assignment.

The `%assign` directive inserts new identifiers into the local function scope (if any), or into the global scope. Identifiers introduced into the function scope are not available within functions being called, and are removed upon return from the function. Identifiers inserted into the global scope are persistent. Existing identifiers can be changed by completely respecifying them. The constant expressions can include any legal identifiers from the `.rtw` files. You can use `%undef` to delete identifiers in the same way that you use it to remove macros.

Within the scope of a function, variable assignments always create new local variables unless you use the :: scope resolution operator. For example, given a local variable foo and a global variable foo:

```
%function …
…
%assign foo = 3
…
%endfunction
```

In this example, the assignment always creates a variable foo local to the function that will disappear when the function exits. Note that foo is created even if a global foo already exists.

In order to create or change values in the global scope, you must use the :: operator to disambiguate, as in:

```
%function …
%assign foo = 3
%assign ::foo = foo
…
%endfunction
```

The :: forces the compiler to assign to the global foo, or to change its existing value to 3.

Note: It is an error to change a value from the RTW 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 %assign directive to create new 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:

```
%assign tempVar = Rec2 {  Name "Name1"; Type "t1" }
%assign Rec1 = Rec1 + Rec2
```

The first statement creates the new Rec2 and the second statement adds the new Rec2 to the existing Rec2. In the first statement, the left-hand side is the reference to the record and the right-hand side is the new record. Figure 2-1 shows the result of adding the record to the existing one:

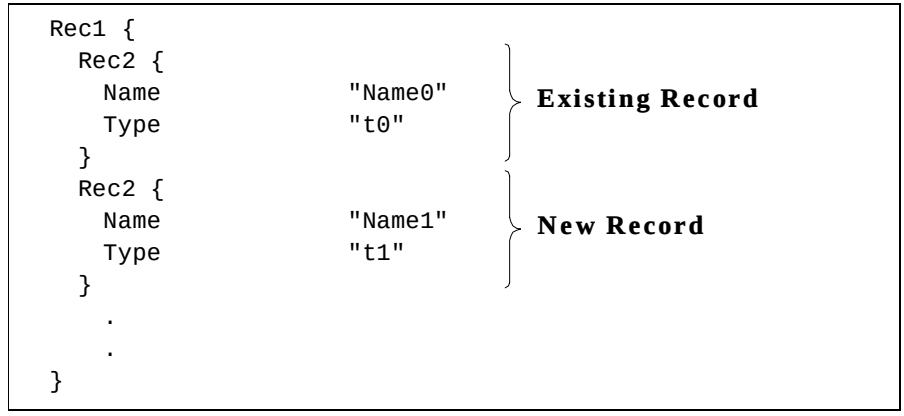


Figure 2-1: Creating a New Record

If you want to access the new record, you can use:

```
%assign myname = tempVar.Name
```

or

```
%assign myname = Rec1.Rec2[1].Name
```

In this same example, if you want to add two records to the existing record, use:

```
%assign tempVar = Rec2 {  Name "Name1"; Type "t1" }
%assign Rec1 = Rec1 + Rec2[0]
%assign tempVar = Rec2 {  Name "Name2"; Type "t2" }
%assign Rec1 = Rec1 + Rec2[1]
```

This produces:

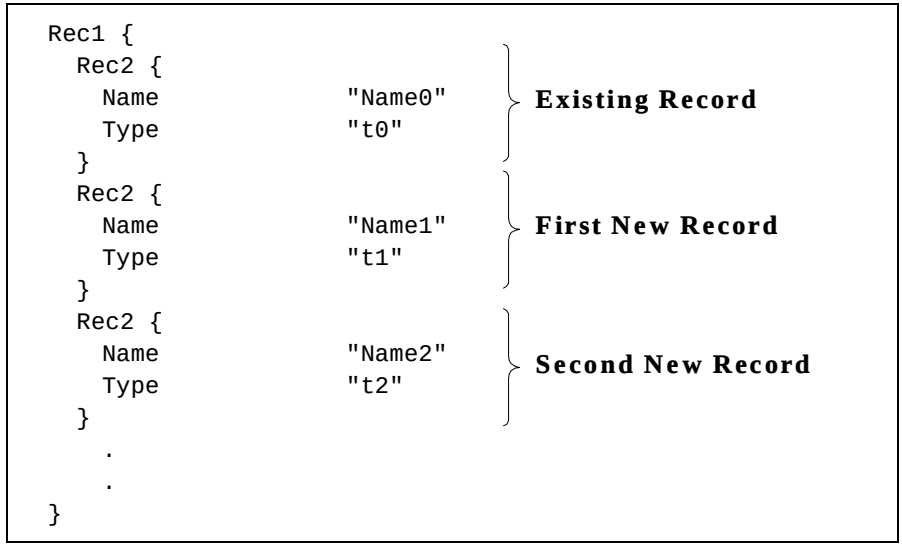


Figure 2-2: Creating Multiple Records

Adding Parameters to an Existing Record

You can use the %assign directive to add a new parameter to an existing record. For example,

```
%assign N = 500
%assign x = Block[Idx] + N    /% 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 Figure 2-3. It returns the block value.

Figure 2-3: Parameter Added to Existing Record

Scoping

The structure of the %with directive is:

```
%with expression
%endwith
```

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

RTW file:

```
System {
   Name              "foo"
}
```

To access the Name parameter without a %with statement, use:

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

or using %with, use:

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

Variable Scoping

The Target Language Compiler uses a form of dynamic scoping to resolve references to variables. This section illustrates the process that the Target Language Compiler performs in determining the values of variables.

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

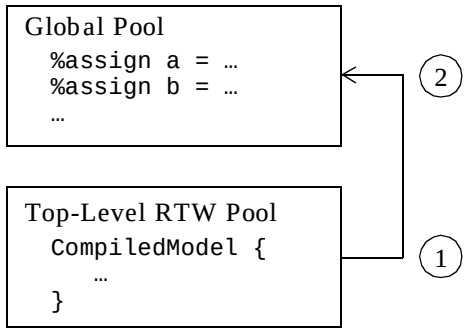


Figure 2-4: Search Sequence

You can modify the search list and search sequence by using the `%with` directive. When you add a construct such as:

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

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

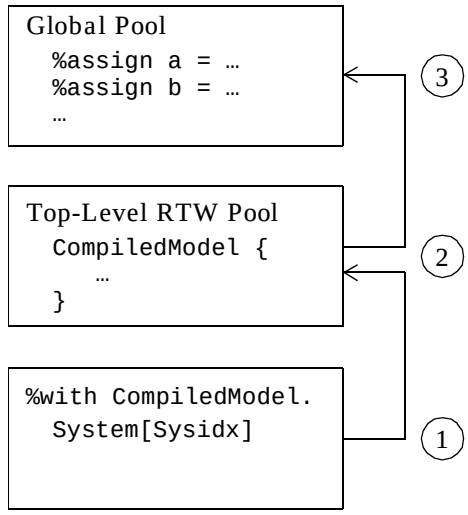


Figure 2-5: Modifying the Search Sequence

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

```
CompiledModel.System[Sysidx].Name
```

Using the pair of `%with` statements as in the previous example, you can refer to the system name simply by:

```
Name
```

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

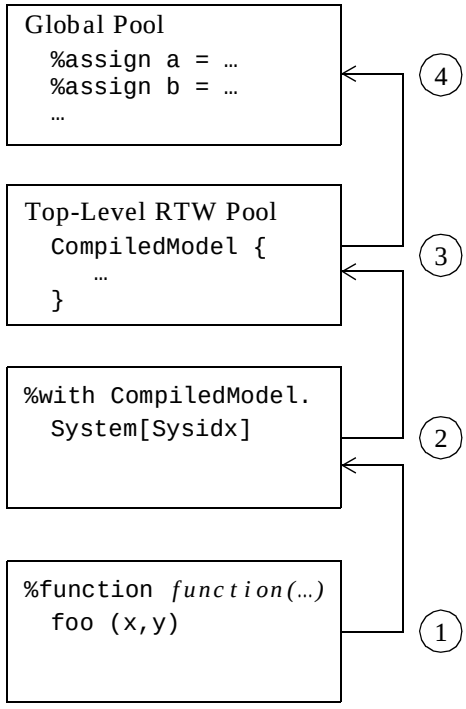


Figure 2-6: Scoping Rules Within Functions

Target Language Functions

The target language function construct is:

```
%function identifier ( optional-arguments ) [Output | void]
%return
%endfunction
```

Functions in the target language are recursive and have their own local variable space. Target language functions do not produce any output, unless they explicitly use the `%openfile`, `%selectfile`, and `%closefile` directives, or are output functions.

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

In this example, a function, name, returns x, if x and y are equal, and returns z, if x and y are not equal.

```
%function name(x,y,z) void

%if x == y
   %return x
%else
   %return z
%endif

%endfunction
```

Function calls can appear in any context where variables are allowed.

All %with statements that are in effect when a function is called are available to the function. Calls to other functions do not include the local scope of the function, but do include any %with statements appearing within the function.

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

By default, a function returns a value and does not produce any output. You can override this behavior by specifying the Output and void modifiers on the function declaration line, as in:

```
%function foo() Output
…
%endfunction
```

In this case, the function continues to produce output to the currently open file, if any, and is not required to return a value. You can use the void modifier to indicate that the function does not return a value, and should not produce any output, as in:

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

Variable Scoping Within Functions

Within a function, the left-hand member of any %assign statement defaults to create a new entry in the function's block within the scope chain, and does not affect any of the other entries. That is, it is local to the function. For example,

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

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

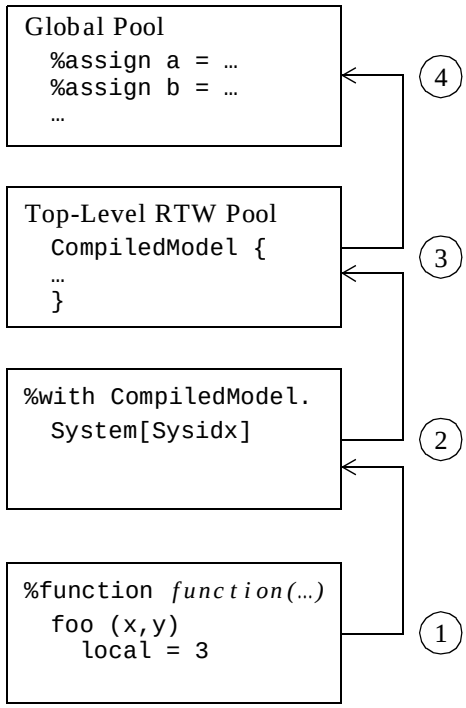


Figure 2-7: Scoping Rules Within Functions Containing Local Variables

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

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

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

When you introduce new scopes within a function using `%with`, these new scopes are used during nested function calls, but the local scope for the function

is not searched. Also, if a `%with` is included within a function, its associated scope is carried with any nested function call. For example,

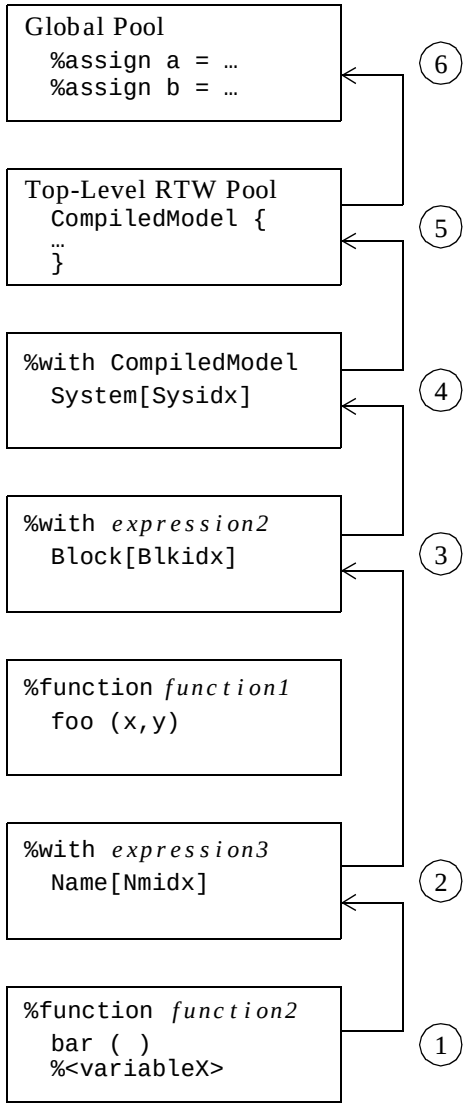


Figure 2-8: Scoping Rules When Using %with Within a Function

%return

The %return statement closes all %with statements appearing within the current function. In this example, the %with statement is automatically closed when the %return statement is encountered, removing the scope from the list of searched scopes.

```
%function foo(s)
    %with s
        %return(name)
    %endwith
%endfunction
```

Target Language Compiler

Command Line Arguments

To call the Target Language Compiler, use:

```
tlc [switch1 expr1 switch2 expr2 ...] filename.tlc
```

Table 2-5 lists the switches you can use with the Target Language Compiler. Order makes no difference.

Table 2-5: Target Language Compiler Switches

Switch	Meaning
`-r` *filename*	Reads a database file (such as a `.rtw` file). Repeat this option multiple times to load multiple database files into the Target Language Compiler. Omit this option for target language programs that do not depend on the database.
`-v[`*number*`]`	Sets the internal verbose level to <*number*>. Omitting this option sets the verbose level to `1`.
`-I`*path*	Adds the specified directory to the list of paths to be searched for `.tlc` files.
`-O`*path*	Specifies any and 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.` (dash Capital O period)
`-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.

Table 2-5: Target Language Compiler Switches (Continued)

Switch	Meaning
`-d[n|g|o]`	Specifies the level and type of debugging. By default, debugging is off (`-do`). `-d` defaults to `-dn`, or normal mode debugging, and `-dg` is generate mode debugging.
`-a`*`ident=expr*	Specifies an initial value for some parameters; equivalent to the `%assign` command. Use this to control template generation by querying its value.

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 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.

Target Language Debug Mode

When you initiate the debug mode via the `-d` switch of the `tlc` command, the Compiler produces a `.log` file for every target file used. The `.log` file contains usage count information regarding how many times each line is encountered during execution.

The output of the listing file includes the number of times each line is encountered followed by a colon.

```
1: %% Abstract: Gain block target file
1:
1: %implements Gain "C"
1:
1: %% Function: FcnEliminateUnnecessaryParams ==================
1: %% Abstract:
1: %%      Elimate unecessary multiplications for following gain
1: %%      cases when inlining 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
1: %%      input
1: %%
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> = %<u>;
0:   %elseif LibIsEqual(k, -1.0)
0:     %<y> = -%<u>;
0:   %else
0:     %<y> = %<u> * %<k>;
0:   %endif
1: %endfunction
1:
1:
1: %% Function: Outputs ========================================
1: %% Abstract:
1: %%      Y = U * K
1: %%
1: %function Outputs(block, system) Output
1:   /* %<Type> Block: %<Name> */
1:   %assign rollVars = ["U", "Y", "P"]
1:   %roll sigIdx = RollRegions, lcv = RollThreshold, block, ...
     "Roller", rollVars
```

```
1:     %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
1:     %assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
1:     %assign k = LibBlockParameter(Gain, "", lcv, sigIdx)
1:     %if InlineParameters == 1
0:       %<FcnEliminateUnnecessaryParams(y, u, k)>\
1:     %else
1:       %<y> = %<u> * %<k>;
1:     %endif
1:   %endroll
1:
1: %endfunction
1:
1: %% [EOF] gain.tlc
```

This structure makes it easy to identify branches not taken and to develop new tests that can exercise unused portions of the target files.

3

Writing Target Language Files

A Basic Example

This section presents an elementary example of creating a target language file that generates specific text from an RTW model. This example shows the sequence of steps that you should follow in creating and using your own target language files.

Process

Figure 3-1 shows the Simulink model, `basic.mdl`.

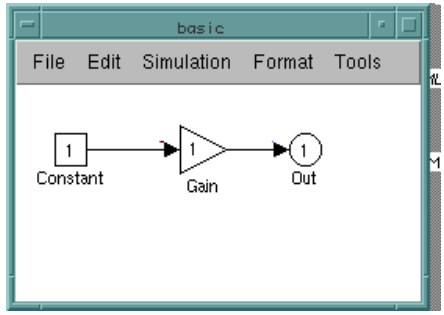


Figure 3-1: Simulink Model

Selecting **Parameters** from Simulink's **Simulation** menu displays the dialog box shown in Figure 3-2.

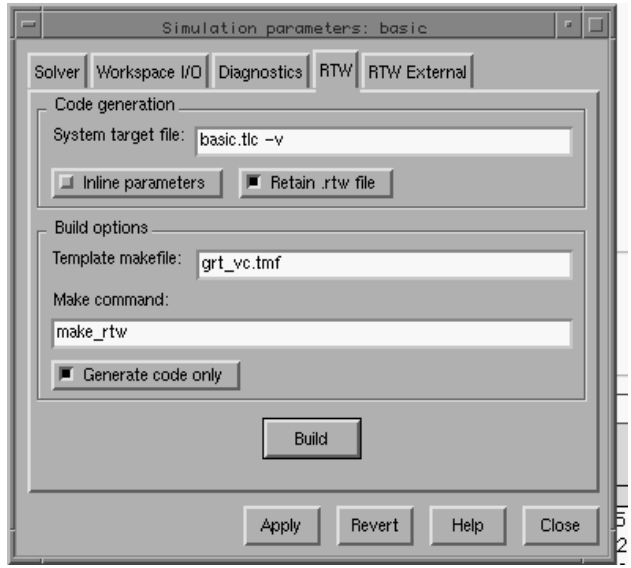


Figure 3-2: Simulation Parameters Dialog

Clicking the **Build** button generates the `.rtw` file, `basic.rtw`, and executes the system target file, `basic.tlc` with the `-v` option, that is specified under **Code generation**. The structure of `basic.rtw` is:

```
CompiledModel {
  Name                        "basic"
  Version                     "1.85 (Oct, 30 1996)"
  GeneratedOn                 "Tue Mar 18 09:53:17 1997"
  Solver                      FixedStepDiscrete
  SolverType                  FixedStep
  StartTime                   0
  StopTime                    10
  FixedStepOpts {
    FixedStep                 0.2
  }
        .
        .
        .
  NumModelInputs              0
  NumModelOutputs             1
  NumNonVirtBlocksInModel     0
  DirectFeedthrough           no
  NumContStates               0
  NumDiscStates               0
        .
        .
        .
BlockOutputs {
    NumBlockOutputs           2
    BlockOutput {
        .
        .
        .
System {
    Type                      root
    Name                      "<root>"
        .
        .
        .
}
```

The file, `basic.tlc`, is a TLC file that uses the `.rtw` code to generate text that contains the model's name, generation date, and its number of continuous states.

`basic.tlc`

```
%with CompiledModel

 My model's name is: %<Name>

 It was generated on: %<GeneratedOn>

 And it has %<NumContStates> continuous states.

%endwith
```

Instead of using the **Simulation Parameters** dialog from Simulink's **Simulation** menu, you could perform the same functions directly from the MATLAB prompt. To create `basic.rtw` and execute `basic.tlc` from the MATLAB prompt, enter:

```
rtwgen basic
tlc –r basic.rtw basic.tlc –v
```

The output of this process is:

```
My model's name is: basic

It was generated on: Tue Mar 18 09:53:17 1997

And it has 0 continuous states.
```

As you continue through this chapter, you will learn the details of creating target files.

Target Language Compiler Architecture

System Target Files

Table 3-1 lists the system target files that generate the RTW code.

Table 3-1: System Target Files

System Target File	Purpose
`grt.tlc`	Assigns specific variables required for generic real-time code generation, and is the Target Language Compiler entry point.
`mdlwide.tlc`	Assigns more generalized variables required for RTW code generation, and is included by `grt.tlc`.
`mdlbody.tlc`	Included by `mdlwide.tlc`.
`mdlreg.tlc`	Included by `mdlbody.tlc`.
`mdlhdr.tlc`	Included by `mdlreg.tlc`.
`mdlparam.tlc`	Included by `mdlreg.tlc`.

Note: The `grt.tlc` file is in the */MATLAB*`/rtw/c/grt` directory. All the system, library, and block target files referred to in this section are in the */MATLAB*`/rtw/c/tlc` directory.

To generate all of the associated generic real-time RTW code, you execute:

```
tlc -r model.rtw MATLAB/rtw/c/grt/grt.tlc -IMATLAB/rtw/c/tlc
```

As the names suggest, `mdlhdr.tlc` creates the model's header file *model*`.h`, `mdlparam.tlc` creates the model's parameters file *model*`.prm`, `mdlreg.tlc` creates the model's function registration file, and `mdlbody.tlc` creates the model's source code file *model*`.c`. If the code size in *model*`.c` reaches the file size threshold, code generation continues in *model1*`.c`. This process continues in *model2*`.c`, and so on.

Program flow through the system target files avoids multiple passes through the RTW file. You should not change the order of code generation without first performing a thorough analysis of the file dependencies. For example, the parameters structure is not created until all source code has been generated. This technique eliminates unused parameters.

Block Functions

The functions declared inside each of the block target files are exercised by the system target files. In these tables, `block` refers to a Simulink block name (e.g., `gain`) and `system` refers to the subsystem in which the block resides.

Table 3-2: Block Functions Exercised by mdlwide.tlc

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

Table 3-3: Block Functions Exercised by mdlbody.tlc

`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 runtime 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.

Coding Conventions

The following guidelines help ensure that the programming style in each TLC file is consistent, and hence, more easily modifiable.

1 All identifiers in the RTW file begin with a capital letter. For example,

```
NumContStates         10
NumBlocks             52
```

Note that block records that contain a `Name` identifier should start the name with a capital letter since the `Name` identifier is often promoted into the parent scope. For example, a block snippet may contain

```
Block {
  :
  TID                 0
  NumRWorkDefines 1
  RWorkDefine {
    Name              "PrevT"
    Width             1
  }
  PrevT               RWorkDefine[0]
  :
}
```

Since the `Name` identifier within the `RWorkDefine` record is promoted to `PrevT` in its parent scope, it must start with a capital 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 TLC assignment directive (%assign) generates a warning if you assign a value to an "unqualified" RTW identifier. For example,

```
%assign TID = 1
```

will produce an error because TID identifier is not qualified by Block. However, a "qualified" assignment will 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.

2. 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 RTW 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
   ```

3. 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]
   ```

4. Library functions (functions in funclib.tlc) start with Lib when the function is to be used outside the library file. If the function is only used

inside the library file, it should start with `Fcn` and the function should be placed at the bottom of `funclib.tlc`.

```
%%
%% Global TLC Functions (start with Lib)
%%

%function LibGlobalTLCFunction(...)

%%
%% Local TLC Functions (start with Fcn)
%%

%function FcnLocalTLCFunction(...)
```

5 Functions declared inside a `block.tlc` file start with `Fcn`. For example,

```
%function FcnMyBlockFunc(...)
```

Note: Functions declared inside a system file are global; functions declared inside a block file are local.

---

6 Do not hard code the variables defined in `mdlwide.tlc`. All RTW global variables start with `rt` and all RTW 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 RTW global variables and functions. These TLC variables are declared in `mdlwide.tlc`.

Table 3-4: TLC Global Variables Specifying RTW Global Variables

Description	TLC Global Variable	Default Value
Block I/O	`tBlockIO`	`rtB`
Block Signal Information	`tModelBlockInfo`	`rtModelBlockInfo`
Control Port Index	`tControlPortIdx`	`controlPortIdx`
Data Store Memory	`tParameters`	`rtP`
External Inputs	`tInput`	`rtU`
External Outputs	`tOutput`	`rtY`
Ground (Unconnected block input)	`tGROUND`	`rtGROUND`
Infinity	`tInf`	`rtInf`
Integer-Work	`tRWork`	`rtRWork`
Minus Infinity	`tMinusInf`	`rtMinusInf`
Mode Vector	`tPWork`	`rtPWork`
Not a Number	`tNan`	`rtNan`
Pointer-Work	`tIWork`	`rtIWork`
Previous Zero-crossing State	`tPrevZCSigState`	`rtPrevZCSigState`
Real-Work	`tChildSimStruct`	`rts`
Root SimStruct	`tDataStores`	`rtDSM`
S-Function SimStruct	`tSimStruct`	`rtS`
States	`tState`	`rtX`
Task Identifier	`tTID`	`tid`

Table 3-5: TLC Global Variables Specifying RTW Global Functions

Description	TLC Global Variable	Default Value
Log variable create function	`tCreateLogVar`	`rt_CreateLogVar`
Log variable update function	`tUpdateLogVar`	`rt_UpdateLogVar`
Zero-crossing function	`tZCFcn`	`rt_ZCFcn`

7 This convention creates consistent variables throughout the TLC files. For example, the Gain block contains the following `Outputs` function:

```
%% Function: Outputs ======================================== (Note 3)
%% Abstract:
%%      Y = U * K
%%
%function Outputs(block, system) Output
 /* %<Type> Block: %<Name> */ ---------- Note 1
 %assign rollVars = ["U", "Y", "P"] ---------- Note 5
 %roll sigIdx = RollRegions, lcv = RollThreshold, block,... (Notes 4, 6)
       "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 2
%endfunction
```

Notes about this TLC code:

Note 1 The code section for each block begins with a comment specifying the block type and name.

Note 2 Include a blank line immediately after the end of the function in order to create consistent spacing between blocks in the output code.

Note 3 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`.

Note 4 For consistency, use the variables `sysIdx` and `blkIdx` for system index and block index, respectively.

Note 5 Use the variable `rollVars` when using the `%roll` construct.

Note 6 Use these conventions to name the loop control variables:

- Use `sigIdx` and `lcv` when looping over `RollRegions`.
- Use `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
```

8 The Target Language Compiler function library files are conditionally included so that they may be included multiple times. For 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_
```

The name of the variable should be the same as the base filename in uppercase with additional underscores attached at both ends.

Writing a Block Target File

To write a block target file, use these polymorphic block functions combined with the Target Language Compiler library functions declared in `funclib.tlc`. For a complete list of the Target Language Compiler library functions, see Chapter 4, "Target Language Compiler Function Library Reference."

A brief description of the necessary block and library functions follow.

TLC Block Setup Functions

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 "Target Language Compiler Function Library Reference" for a list of relevant functions to call from inside this block function. See `fromwks.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

As an example, given S-function `foo` with a scalar parameter representing a `gain` and one `RWork` representing previous inputs, you could define the following function.

```
%function BlockInstanceSetup(block, system) void

  %% Rename S-function parameter so that the TLC and
  %% generated code is more readable (P1 --> Gain)

  %<LibRenameParameter(block, P1, "Gain")>

  %% Define the RWork vector to make the code more
  %% readable, and to allow the RWork to be rolled
  %% inside the %roll construct.

  %<LibDefineRWork(block, "PrevU", LibDataInputPortWidth(0))>

%endfunction
```

Now you can reference `P1` as `PrevU` in the Target Language Compiler. For example,

```
%function Outputs(block, system) Output
 %assign PrevU = LibBlockParameter(PrevU, ucv, lcv, sigIdx)
  .
  .
  .
%endfunction
```

And, the generated code produces `"rtP.foo.PrevU"` instead of `"rtP.foo.P1"`.

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 4, "Target Language Compiler 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 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
```

TLC Output Block Functions

The remaining block functions are Target Language Compiler output functions executed by the system target file `mdlbody.tlc` for each block in the model.

Enable(block, system)

Nonvirtual subsystem `Enable` functions are created whenever a Simulink subsystem contains an Enable block. Including the `Enable` function in a block's TLC file places the block's specific enable code into this subsystem `Enable` function. See `sin_wave.tlc` for an example of the `Enable` function.

Disable(block, system)

Nonvirtual subsystem `Disable` functions are created whenever a Simulink subsystem contains a Disable block. Including the `Disable` function in a block's TLC file places the block's specific disable code into this subsystem `Disable` function. See `outport.tlc` for an example of the `Disable` function.

Start(block, system)

Include a `Start` function to place code into `MdlStart`. The code inside `MdlStart` executes once and only once. Typically, you include a `Start` function to execute code once at the beginning of the simulation (e.g., initialize values in the work vectors; see `backlash.tlc`,) or code that does not need to be re-executed when the subsystem in which it resides enables. See `constant.tlc` for an example of the `Start` function.

InitializeConditions(block, system)

TLC code that is generated from the block's `InitializeConditions` function ends up in one of two places. The code is placed into `MdlStart` if the Simulink block does not reside in a nonvirtual subsystem that requires an `Initialize` function. That is, a nonvirtual subsystem contains an `Initialize` function when it is configured to reset states on enable. If this is the case, the TLC code generated from this block function is placed in the subsystem `Initialize` function, and `MdlStart` will call this `Initialize` function. However, if the Simulink block resides in root or a nonvirtual subsystem that does not require an `Initialize` function, the code generated from this block function is placed directly (inlined) into `MdlStart`.

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, and you include an `InitializeConditions` function to execute code that must re-execute when the subsystem in which it resides enables. See `delay.tlc` for an example of the `InitializeConditions` function.

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 `MdlOutputs` if the Simulink block does not reside in a nonvirtual subsystem. The code is placed in a subsystem's `Outputs` function if the Simulink block resides in a nonvirtual subsystem. See `gain.tlc` for an example of the `Outputs` function.

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 each major time step. Code generated from this function is either placed into `MdlUpdate` 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.

Derivatives(block, system)

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

The RTW TLC Function Library

The file `funclib.tlc` contains the RTW TLC function library. This file contains the necessary TLC functions required to write a block target file.

Chapter 4, "Target Language Compiler Function Library Reference," contains detailed descriptions of all the TLC functions. This section focuses on the most commonly used TLC functions, providing a general description of the functions.

Table 3-6: Common TLC Functions

`LibDefineRWork(block, name, width)`
`LibDefineIWork(block, name, width)`
`LibDefinePWork(block, name, width)`
`LibCacheFunctionPrototype(buffer)`
`LibCacheDefine(buffer)`
`LibIsDiscrete(tid)`
`LibDataOutputPortWidth(portIdx)`
`LibDataInputPortWidth(portIdx)`
`LibBlockOutputSignal(portIdx, ucv, lcv, sigIdx)`
`LibBlockInputSignal(portIdx, ucv, lcv, sigIdx)`
`LibBlockParameter(paramRef, ucv, lcv, sigIdx)`
`LibBlockParameterAddr(paramRef, ucv, lcv, sigIdx)`
`LibBlockMatrixParameter(paramRef, rowUcv, rowLcv, rowSigIdx, colUcv, colLcv, colSigIdx)`
`LibBlockMatrixParameterAddr(paramRef, ucv, lcv, sigIdx)`
`LibDiscreteState(ucv, lcv, sigIdx)`
`LibConinuousState(ucv, lcv, sigIdx)`
`LibBlockMode(ucv, lcv, sigIdx)`
`LibBlockRWork(rworkRef, ucv, lcv, sigIdx)`

Table 3-6: Common TLC Functions (Continued)

`LibBlockIWork(iworkRef, ucv, lcv, sigIdx)`
`LibBlockPWork(pworkRef, ucv, lcv, sigIdx)`
`LibCacheNonFiniteAssignment(assignment)`
`LibPrevZCState(ucv, lcv, sigIdx)`
`LibDataStoreMemory(ucv, lcv, varIdx)`
`LibPathName(name)`
`LibIsFinite(value)`
`LibRenameParameter(block, param, newName)`
`LibBlockOutportLocation(ucv, lcv, sigIdx)`

LibDefineRWork(block, name, width)

This call should be made from inside the block's `BlockInstanceSetup` function, and adds the specified `RWork` definition to the block. The function creates and maintains an internal record for the `RWork` definition, removing the Simulink definition if necessary.

LibDefineIWork(block, name, width)

This call should be made from inside the block's `BlockInstanceSetup` function, and adds the specified `IWork` to the block. The function creates and maintains an internal record for the `IWork` definition. For example, a block may have `IWork` records for system enable.

LibDefinePWork(block, name, width)

This call should be made from inside the block's `BlockInstanceSetup` function, and adds the specified `RWork` definition to the block. The function creates and maintains an internal record for the `RWork` definition, removing the Simulink definition if necessary.

LibCacheFunctionPrototype(buffer)

This function 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 among other generated function prototypes.

LibCacheDefine(buffer)

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 among other generated #define statements.

LibIsDiscrete(tid)

Based on the block's TID, this function returns 1 if the block is discrete, otherwise, it returns 0. For an example of this function, see sin_wave.tlc.

LibDataOutputPortWidth(portIdx)

Based on the output port index, this function returns the width of the data port. For an example of this function, see css.tlc.

LibDataInputPortWidth(portIdx)

Based on the input port index, this function returns the width of the data port. For an example of this function, see css.tlc.

LibBlockOutputSignal(portIdx, ucv, lcv, sigIdx)

Based on the output port index (portIdx), the user control variable (ucv), the loop control variable (lcv), and the signal index (sigIdx), this function returns a reference to the block I/O data structure. For example,

Case	Function	May Produce
1	LibBlockOutputSignal(0, "i", "", sigIdx)	rtB.blockname[i]
2	LibBlockOutputSignal(0, "", lcv, sigIdx)	y0[i1]
3	LibBlockOutputSignal(0, "", lcv, sigIdx)	rtB.blockname[0]

Given the same set of arguments, this function returns the appropriate reference to the block's output signal depending on the state of RTW code generation. For example, case 1 is generated since the user control variable is specified. Cases 1 and 2 receive the same arguments, lcv and sigIdx, however,

they generate different results depending on whether RTW is in a loop-rolling state, or a non loop-rolling state, respectively.

Loop rolling is fully described later in this chapter. In short, however, this function looks at `ucv`, `lcv`, and `sigIdx`, and the RTW state 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. If `ucv` is not specified and `lcv` and `sigIdx` are specified, the returned value depends on whether or not RTW is currently rolling. If RTW is currently in a loop rolling state, `lcv` is used, otherwise `sigIdx` is used. If neither `ucv` or `lcv` are not specified, `sigIdx` is used. For an example of this function, see `gain.tlc`.

LibBlockInputSignal(portIdx, ucv, lcv, sigIdx)

The behavior of this function is similar to `LibBlockOutputSignal`, except it returns the appropriate reference to a block's input signal. For example,

Function	May Produce
`LibBlockInputSignal(0, "i", "", sigIdx)`	`rtB.blockname[i]`
`LibBlockInputSignal(0, "", lcv, sigIdx)`	`u0[i1]`
`LibBlockInputSignal(0, "", lcv, sigIdx)`	`rtB.blockname[0]`

For an example of this function, see `gain.tlc`.

LibBlockParameter(param, ucv, lcv, sigIdx)

The behavior of this function is similar to `LibBlockOutputSignal`, except it returns the appropriate reference to a block's parameter. The function can only be used for parameters of type `Scalar` or `Vector`. For example,

Function	May Produce
`LibBlockParameter(Gain, "i", "", sigIdx)`	`rtP.Gain[i]`
`LibBlockParameter(Gain, "", lcv, sigIdx)`	`p_Gain[i1]`
`LibBlockParameter(Gain, "", lcv, sigIdx)`	`rtP.Gain[0]`
`LibBlockParameter(Gain, "", lcv, sigIdx)`	`2.5`

For an example of this function, see `gain.tlc`.

Note 1: Do *not* use this function to build the address of a parameter. For example,

```
%assign paramAddr = "&%<LibBlockParameter(Gain, ...)>"
```

This may produce a reference to a constant number, for example `&4.95` if the value of `Gain` is 4.95, and RTW is configured to inline parameter values. Use `LibBlockParameterAddr` to avoid this undesirable behavior.

Note 2: Code generation exits if this function is passed a matrix parameter. (see `LibBlockMatrixParameter`).

LibBlockParameterAddr(param, ucv, lcv, sigIdx)

This function returns the appropriate address of a block's parameter. The function works similarly to `LibBlockParameter` except that its returned value is independent of the inline parameter values configuration. That is, `LibBlockParameterAddr(Gain, "i", "", sigIdx)` will return `&rtP.Gain[i]` regardless if RTW is configured to inline parameter values. For an example of this function, see `lookup2d.tlc`.

Note: Calling this function will force the parameter to stay in memory regardless of the value of `InlineParameters`.

LibBlockMatrixParameter(param, rowUcv, rowLcv, rowSigIdx, colUcv, colLcv, colSigIdx)

This function is similar to `LibBlockOutputSignal`, except it returns the appropriate reference to a block's matrix parameter. These are user control variables, loop control variables, and signal indices for both the rows and

columns of your matrix. This function will degenerate to a vector or scalar, if needed.

Note: Do *not* use this function to build the address of a parameter. For example,

```
%assign paramAddr = "&%<LibBlockMatrixParameter(TruthTable, ...)>"
```

This may produce a reference to a constant number, for example, `&4.95` f the value of `TruthTable` is 4.95, and RTW is configured to inline parameter values. Instead, use the TLC function `LibBlockParameterAddr` to avoid this undesirable behavior. For an example of this function, see `cmblogic.tlc`.

LibBlockMatrixParameterAddr(param, rowUcv, rowLcv, rowSigIdx, colUcv, colLcv, colSigIdx)

This function returns the appropriate address of a block's matrix parameter. The function is similar to `LibBlockParameterAddr`. For an example of this function, see `cmblogic.tlc`.

Note: Calling this function will force the parameter to stay in memory regardless of the value of `InlineParameters`.

LibDiscreteState(ucv, lcv, sigIdx)
LibContinuousState(ucv, lcv, sigIdx)

The behavior of these functions is similar to `LibBlockOutputSignal`, except they return the appropriate reference to the block's discrete/continuous state. For an example of these functions, see `delay.tlc` and `integrat.tlc`.

LibBlockMode(ucv, lcv, sigIdx)

The behavior of this function is similar to `LibBlockOutputSignal`, except it returns the appropriate reference to a block's mode. For an example, see `dintegrt.tlc`.

LibBlockRWork(rworkRef, ucv, lcv, sigIdx)
LibBlockIWork(iworkRef, ucv, lcv, sigIdx)
LibBlockPWork(pworkRef, ucv, lcv, sigIdx)

The behavior of these functions is similar to `LibBlockOutputSignal`, except they return the appropriate reference to the block's `RWork`, `IWork`, and `PWork`. The additional arguments, `rworkRef`, `iworkRef`, and `pworkRef`, are references to the block internal records `RWorkDefine`, `IWorkDefine`, and `PWorkDefine`, respectively. For an example of `LibBlockRWork` and `LibBlockIWork`, see `sin_wave.tlc`. For an example of `LibBlockPWork`, see `towks.tlc`.

If the block records `RWorkDefine`, `IWorkDefine`, and `PWorkDefine` are not defined, then the reference to the work records is replaced with a reference to the block's vector identifier, `RWork`, `IWork`, and `PWork`, respectively.

LibPrevZCState(ucv, lcv, sigIdx)

The behavior of this function is similar to `LibBlockOutputSignal`, except it returns the appropriate reference to a block's previous zero-crossing state. For an example, see `subsystm.tlc`.

LibDataStoreMemory(ucv, lcv, varIdx)

The behavior of this function is similar to `LibBlockOutputSignal`, except it returns the appropriate reference to a block's data store memory element. For an example, see `dsread.tlc`.

LibPathName(name)

Given a block or system name, this function returns the full path. For example, `LibPathName("<s5>/foo")` may return `root_foosystem_fooblock`, meaning that the block's name is derived from a block named `fooblock` residing in a subsystem block named `foosystem`, residing in the `root` model. For an example of this function, see `mdlbody.tlc`.

LibIsFinite(value)

Given a TLC variable, this function returns 0 if the value of the variable is `rtInf`, `rtMinusInf`, or `rtNaN`. For an example of this function, see `mdlparam.tlc`.

LibRenameParameter(block, param, newName)

Given a reference to a block, a reference to a block parameter, and the new name for the block parameter, this function renames the parameter and creates a new reference to the parameter. Most likely, you will call this function from `BlockInstanceSetup`.

As an example, if you want to rename the S-function parameter `P1` defined in this RTW file:

```
System {
  Block {
    Name "sfunc"
    Type "S-Function"
    :
    Parameter {
      Name "P1Size"
      Value [1, 1]
      String ""
    }
    Parameter {
      Name "P1"
      Value 2.5
      String "Kp"
    }
    :
  }
}
```

Call `LibRenameParameter(Block, Parameter[1], "Kp")`, which renames `P1` to `Kp` and creates a `Kp` identifier that references the `Parameter[1]`. The block record becomes:

```
System {
  Block {
    Name "sfunc"
    Type "S-Function"
    :
    Parameter {
      Name "P1Size"
      Value [1, 1]
      String ""
    }
    Parameter {
      Name "Kp"
      Value 2.5
      String "Kp"
    }
    Kp Parameter[1]
    :
  }
}
```

Note: By convention, start parameter names with a capital letter since the `Name` identifier of the `Parameter` record is promoted into the parent block scope. It is not mandatory that you do so, however, the Target Language Compiler will exit if you attempt this assignment:

```
%assign kp = LibBlockParameter(kp, "", "", 0)
```

The Target Language Compiler exits because it does not know if the first `kp` is the block identifier `kp` or a local variable `kp`. One way to avoid the confusion is to qualify which `kp` you are assigning. A valid assignment is:

```
%assign block.kp = LibBlockParameter(kp, "", "", 0)
```

Avoid the confusion and stick to the convention by renaming the variable `Kp` in the TLC file

```
%assign kp = LibBlockParameter(Kp, "", "", 0)
```

LibBlockOutportLocation(ucv, lcv, sigIdx)

The behavior of this function is similar to `LibBlockOutputSignal`, except it returns the appropriate reference to a block outport signal. For an example, see `outport.tlc`.

LibCacheNonFiniteAssignment(assignment)

This function should be called from inside `BlockTypeSetup` to cache assignments that need to be placed in the registration function because of nonfinite initialization. That is, the `rtInfs`, `rtNaNs`, and `rtMinusInfs` parameters are initialized to zero until the registration function is called, re-initializing them to their appropriate value. Each call to this function appends your buffer to the existing cache buffer.

Built-In TLC Functions

The most common built-in TLC functions required to write a block target file are `STRINGOF`, `EXISTS`, and `SIZE`.

STRINGOF(value)

Given an RTW string vector, this function returns the reconstructed string. For example, this function returns the string `"float"`.

```
%<STRINGOF([102, 108, 111, 97, 116])>
```

The built-in function `SIZEOF` is commonly used to reconstruct S-function parameters that are literal strings. For an example of this function, see *MATLAB_ROOT*`/rtw/c/mwdspace/devices/dp_read.tlc`.

EXISTS("name")

This built-in function determines if `name` exists in the current scope space. Note that `EXISTS` commands search the current scope backwards to the root scope.

SIZE(value, n)

The behavior of this built-in function is:

If n =	This Function Returns
`0`	The number of rows in `value`.
`1`	The number of columns in `value`.
`2`	`[nRows, nCols]` in `value`.

Inlining an S-Function

When a Simulink model contains an S-function and a corresponding `.tlc` file, Real-Time Workshop inlines the S-function. Inlining an S-function can produce more efficient code by eliminating the S-function API layer from the generated code.

S-functions that are not inlined make calls to all of these seven functions, even if the routine is empty for the particular S-function:

S-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.

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.

An 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 S-function is `foogain`, the C-coded MEX-file must contain:

```
#define S_FUNCTION_NAME foogain
#include "simstruc.h"
#define GAIN mxGetPr(ssGetArg(S,0))[0]

static void mdlInitializeSizes(SimStruct *S)
{
  ssSetNumContStates     (S, 0);
  ssSetNumDiscStates     (S, 0);
  ssSetNumInputs         (S, 1);
  ssSetNumOutputs        (S, 1);
  ssSetNumInputArgs      (S, 1);
  ssSetDirectFeedThrough (S, 1);
  ssSetNumSampleTimes    (S, 0);
  ssSetNumIWork          (S, 0);
  ssSetNumRWork          (S, 0);
  ssSetNumPWork          (S, 0);
}

static void
mdlOutputs(real_T *y, const real_T *x, const real_T *u,
           SimStruct *S, int_T tid)
```

```
{
  y[0] = u[0] * GAIN;
}

static void
mdlInitializeSampleTimes(SimStruct *S){}

static void
mdlInitializeConditions(real_T *x0,SimStruct *S) {}

static void
mdlUpdate(real_T *x, const real_T *u,SimStruct *S, int_T tid) {}

static void
mdlDerivatives(real_T *dx, const real_T *x const real_T *u,
               SimStruct *S, int_T tid) {}

static void
mdlTerminate(SimStruct *S) {}

#ifdef  MATLAB_MEX_FILE
#include "simulink.c"
#else
#include "cg_sfun.h"
#endif
```

Without a TLC file to define the S-function specifics, RTW must call the MEX-file S-function in a manner similar to RTW 1.3. That is, the execution of the S-function is through the S-function API.

```
void
MdlStart()
{
  /* S-Function block: foo */
  {
   SimStruct *s = ssGetSFunction(S, 0);
    real_T *sfcnX = ssGetX(s);

   sfcnInitializeConditions(sfcnX, s);
  }
}
```

Unnecessary call to empty function `mdlInitializeConditions` in `foogain.c`.

```
void
MdlOutputs(tid)
{
  /* S-Function block: foo */
  {
    Simstruct *s = ssGetSFunction(S, 0);
    real_T *sfcnU = ssGetU(s);
    real_T *sfcnX = ssGetX(s);
    real_T *sfcnY = ssGetY(s);

    sfcnOutputs(sfcnY, sfcnX, sfcnU, s, tid);
  }
}

void
MdlUpdate(tid)
{
  /* S-Function block: foo */
  {
   Simstruct *s = ssGetSFunction(S, 0);
    real_T *sfcnX = ssGetX(s);
    real_T *sfcnU = ssGetU(s);

    sfcnUpdate(sfcnX, sfcnU, s, tid);
  }
}
```

Unnecessary call to empty function `mdlUpdate` in `foogain.c`.

```
void
MdlDerivatives()
{
  /* S-Function block: foo */
  {
   Simstruct *s = ssGetSFunction(S, 0);
    real_T *sfcnU = ssGetU(s);
    real_T *sfcnX = ssGetX(s);
    real_T *sfcndX = ssGetdX(s);

    sfcnDerivatives(sfcndX, sfcnX,
                    sfcnU, s, tid);
  }
}
```

Unnecessary call to empty function `mdlDerivatives` in `foogain.c`.

```
void
MdlTerminate()
{
  /* S-Function block: foo */
  {
   Simstruct *s = ssGetSFunction(S, 0);
    sfcnTerminate(s);
  }
}
```

Unnecessary call to empty function `mdlTerminate` in `foogain.c`.

```
/* function to register model in
SimStruct */
Simstruct *
foogain()
{
  :
  :/* Normal model initialization code independent of
      S-functions  */
  :

/* S-function initialization code required for all S-functions
   without corresponding TLC files */

/* set number of children S-Functions */
  ssSetNumSFunctions(S, 1);
```

Note 3

```
/* Register children S-Functions (s-funcs without TLC files) */
{
  static SimStruct childSFunctions[1];
  static SimStruct *childSFunctionPtrs[1];
  ssSetSFunctions(S, (SimStruct **) &childSFunctionPtrs[0]);

  /* S-Function Block: foo */
  {
    static real_T sfcnPeriod[1];
    static real_T sfcnOffset[1];
    static real_T sfcnTsMap[1];
    static mxArray *sfcnParams[1];
    extern void foogain(SimStruct *);
    SimStruct *s = &childrenSFunctions[0];

    memset((char *) s, 0, sizeof(SimStruct));

    ssSetModelName(s, "foogain");
    ssSetPath(s, "foogain");
    ssSetParentSS(s, S);
    ssSetRootSS(s, ssGetRootSS(S));
    ssSetSFcnParamsCount(s, 1);
    ssSetSFcnParamsPtr(s, (const mxArray **) &sfcnParams[0]);
    ssSetSFcnParam(s, 0, (real_T *) &P.foo.P1Sizes[0]);
    ssSetU(s, &rtGround);
    ssSetY(s, &B.foo);
    ssSetMdlInfoPtr(s, ssGetMdlInfoPtr(S));
    ssSetSampleTimePtr(s, (real_T *) &sfcnPeriod[0]);
    ssSetOffsetTimePtr(s, (real_T *) &sfcnOffset[0]);
    ssSetSampleTimeTaskIDPtr(s, (int_T *) &sfcnTsMap[0]);

    sfunctionName(s);

    ssSetSFunction(S, 0, s);
    sfcnInitializeSizes(s);
    sfcnInitializeSampleTimes(s);
  }
 }
}
```

To avoid unnecessary calls to the S-function and to generate the minimum code required for the S-function, the following TLC file is provided as an example.

foogain.tlc

```
%implements "foogain" "C"    ————— Note 1

%function BlockInstanceSetup(block, system) void
 %<LibRenameParameter(block, P1, "Gain")>
%endfunction

%function Outputs(block, system) Output
  %assign y = LibBlockOutputSignal(0, "", "", 0)
  %assign u = LibBlockInputSignal(0, "", "", 0)
  %assign p = LibBlockParameter(Gain, "", "", 0)
  /* %<Type> block: %<Name> */
  %<y> = %<u> * %<p>;
%endfunction
```

By including this simple target file for this S-function block, the code is generated as:

```
void
MdlStart(){}

void
MdlOutputs(tid)
{
  /* S-Function block: foo */
  rtB.foo = rtGROUND * rtP.foo.gain;
}

void
MdlUpdate(tid){}

void
MdlTerminate(){}

/* function to register model in SimStruct */
Simstruct *foogain()
{
  :
  : /* Model Registration
       (does NOT need to register S-function) */
  :
}
```

Note 2

Note 3

So, 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:

Note 1 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`.

Note 2 The input to `foo` is `rtGROUND` (an RTW global equal to 0.0) since `foo` is the only block in the model, and its input is unconnected. Had it been connected to block `sinewave`, the generated line would have been

```
rtB.foo = rtB.sinewave * rtP.foo.gain;
```

Note 3 Including a TLC file for `foogain` eliminated the need for an S-function registration segment for `foogain`. This significantly reduces code size.

Note 4 The TLC code will inline the gain parameter when RTW 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;
```

Note 5 Use the `%generatefile` directive if your operating system has filename size restriction and the name of the S-function is `foosfunction` (which 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`.

Configurable RTW Variables

This table lists the configurable RTW variables.

Table 3-7: Configurable RTW Variables

Variable	Purpose
`InlineParameters`	Inlines parameter values in the generated code. Possible values are 0 or 1 with a default of 0.
`FileSizeThreshold`	Specifies the maximum number of lines to output to *model*`.c` before the file is split into `model1.c`, `model2.c`, etc. Default value is 50,000 lines.
`MatFileLogging`	Creates a MATLAB workspace containing the output of Outports, Scopes, and To Workspace blocks. Possible values are 0 or 1 with a default of 1.
`ModelBlockInfo`	Creates a data structure for independently monitoring signals in the model. Possible values are 0 or 1 with a default of 0.
`RollThreshold`	Specifies the threshold for loop rolling. The `%roll` directive uses this value to determine whether a section of code should be enclosed into a `for` loop.

These variables can be changed at the command line. For example,

```
tlc -r model.rtw MATLAB/rtw/c/grt/grt.tlc
    -IMATLAB/rtw/c/tlc -aInlineParameters=1
```

Or, if you're using the RTW GUI within Simulink, simply modify the system target file dialog entry box. For example, to override the default behavior of `FileSizeThreshold`, modify the dialog box as:

```
grt.tlc -aFileSizeThreshold=20000
```

Figure 3-3 shows the RTW dialog box with the change to modify the `FileSizeThreshold`.

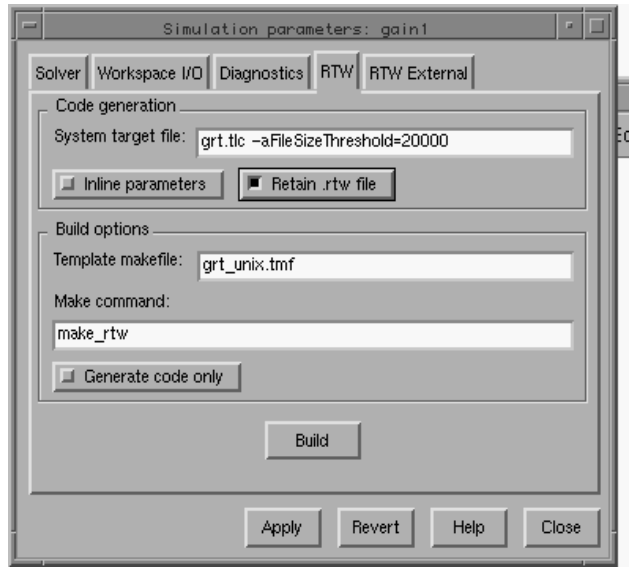


Figure 3-3: RTW Dialog Box

Matrix Parameters in RTW

MATLAB matrices are the transpose of RTW matrices, with the exception of S-function blocks, which use the MATLAB representation. MATLAB uses column-major ordering and RTW uses row-major ordering for everything except S-function blocks. The Target Language Compiler follows this behavior to ensure backward compatibility.

The Target Language Compiler declares all Simulink block parameters as

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

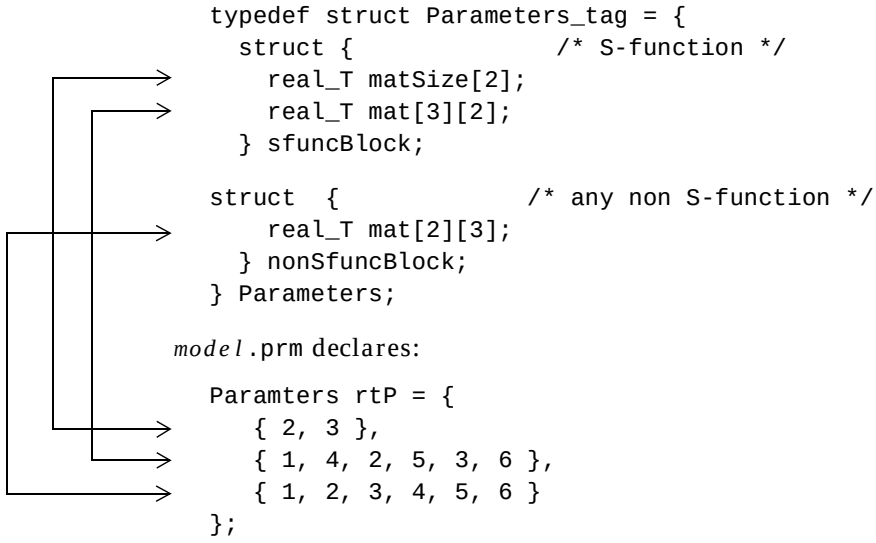
with the exception of S-function blocks, which are declared as

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

For example, given the 2-by-3 matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

model.h defines:



```
typedef struct Parameters_tag = {
  struct {              /* S-function */
    real_T matSize[2];
    real_T mat[3][2];
  } sfuncBlock;

struct  {             /* any non S-function */
    real_T mat[2][3];
  } nonSfuncBlock;
} Parameters;
```

model.prm declares:

```
Paramters rtP = {
   { 2, 3 },
   { 1, 4, 2, 5, 3, 6 },
   { 1, 2, 3, 4, 5, 6 }
};
```

The Target Language Compiler access routines, `LibBlockMatrixParameter` and `LibBlockMatrixParameterAddr`, return:

```
LibBlockMatrixParameter(mat, "", "", 0, "", "", 1) returns
```

non-S-function: 2
S-function: 2

```
LibBlockMatrixParameterAddr(mat, "", "", 0, "", "", 1) returns
```

non-S-function: `&rtP.nonSfuncBlock[0][1];`
S-function: `&rtP.sfuncBlock[1][0];`

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, `matSize` is not declared

unless it is explicitly accessed by `LibBlockParameter` or `LibBlockParameterAddr`.

Loop Rolling

The best way to explain loop rolling is by example. Figure 3-4 shows a Simulink model with a Gain block.

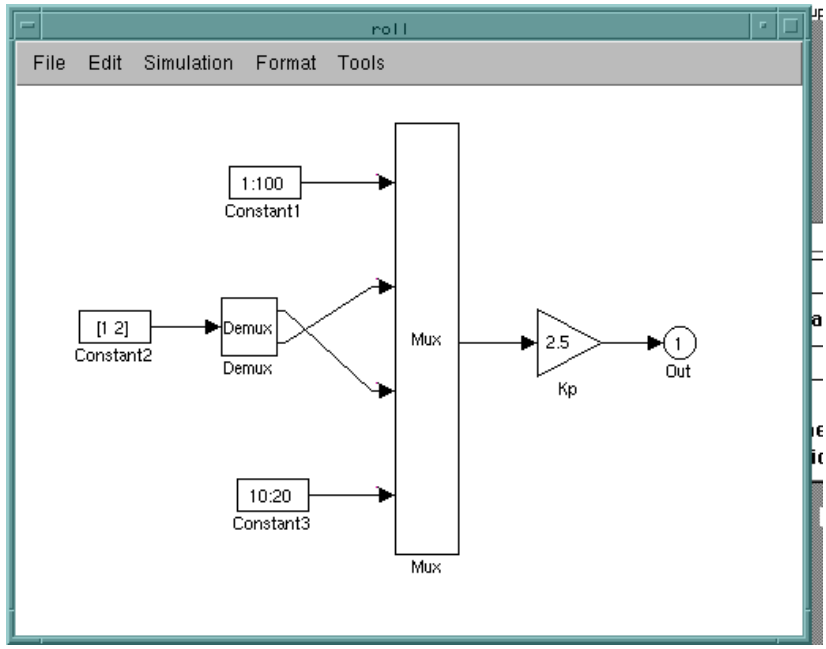


Figure 3-4: Example of Loop Rolling

The outputs function for the Gain block is:

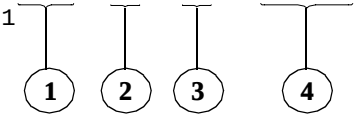
```
%% Function: Outputs ========================================
%% Abstract:
%%      Y = U * K
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign rollVars = ["U", "Y", "P"]
  %roll sigIdx = RollRegions, lcv = RollThreshold, block,...
        "Roller", rollVars
    %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
    %assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
    %assign k = LibBlockParameter(Gain, "", lcv, sigIdx)
      %<y> = %<u> * %<k>;
  %endroll

%endfunction
```

The generated code will roll depending on the block's `RollRegion` (specified in the RTW file) and `RollThreshold` (specified at the command line). If there are any regions in `RollRegion` that are greater than the value specified by `RollThreshold`, then those regions will roll. However, any regions in `RollRegion` that are less that the value specified by `RollThreshold` will be expanded.

For example, the Gain block in this example is defined in the RTW file as:

```
Block {
  Type                Gain
  Name                "<Root>/Kp"
  Identifier          Kp
  TID                 0
  RollRegions         [0:99, 100, 101, 102:112]
  NumDataInputPorts   1
      .
      .
      .
  Parameter {
    Name              "Gain"
    Value             [2.5]
    String            "2.5"
  }
}
```



Note that there are four `RollRegions` in this example:

- One contiguous region for `Constant1` inputs (1)
- Two noncontiguous regions for `Constant2` inputs (2 and 3)
- One contiguous region for `Constant3` (4)

The generated code for this example is:

```
/* Gain Block: <Root>/Kp */
  {
    int_T i1;
    real_T *u0 = &rtB.Constant1[0];
    real_T *y0 = &rtB.Kp[0];

    for(i1 = 0; i1 < 100; i1++) {
      y0[i1] = u0[i1] * rtP.Kp.Gain;
    }

    rtB.Kp[100] = rtB.Constant2[1] * rtP.Kp.Gain;
    rtB.Kp[101] = rtB.Constant2[0] * rtP.Kp.Gain;

    u0 = &rtB.Constant3[0];
    y0 = &rtB.Kp[102];

    for(i1 = 0; i1 < 11; i1++) {
      y0[i1] = u0[i1] * rtP.Kp.Gain;
    }
  }
```

RollRegion 1

RollRegions 2 & 3

RollRegion 4

Note that `%roll` requires `rollVars` to be specified. The `rollVars` variable tells the loop roller which variables to set up within the roll scope. Note that in this case `P` was not declared despite the fact it was specified. This is because it is a scalar value, hence, it need not be declared.

As you can see the `%roll` degenerates to `%foreach` when the code doesn't roll. Thus, you should write the TLC code assuming the `%foreach` case. That is, don't special case your code to handle both cases, rather, write the code once

with the %roll that works under both situations. Table 3-8 contains the valid variables assigned to rollVars.

Table 3-8: Roll Table Variables

Block	Variable	Description
Inputs	U u*i*	All inputs input *i*
Outputs	Y y*i*	All outputs output *i*
Parameters	P <param>/name	All parameters parameter name
RWork	RWork <rwork>/name	All RWorks name rwork
IWork	IWork <iwork>/name	All IWorks name iwork
PWork	PWork <pwork>/name	All PWorks name pwork
Mode	M	Mode
Previous Zero-Crossing	PZC	Zero-crossings

For example,

```
%assign rolVars = ["u0" "RWork" "<param>/Gain"]
%roll SigIdx = lcv = RollThreshold, block, "Roller", rollVars
```

declares the first block input (input zero), all the block's RWorks, and the Block parameter, Gain.

4

Target Language Compiler Function Library Reference

LibBlockFunctionExists

Purpose

Determines if a given block function exists

Syntax

```
%<LibBlockFunctionExists(block, fcn)>
```

Arguments

`block`
Reference to a block record.

`fcn`
Function to check (e.g., "`Outputs`").

Returns

Value	Condition
`1`	Specified function exists.
`0`	Specified function does not exist, but TLC file does.
`-1`	TLC file does not exist.

Description

Determines if a given block function exists. `LibBlockFunctionExists` first checks to see if the TLC file exists (for S-function blocks). Then it checks to see if the function exists.

Purpose

Determines the input signal label based on the type of input signal

Syntax

```
%<LibBlockInputSignal(portNum, ucv, lcv, sigIdx)>
```

Arguments

`portNum`
Integer data input port number.

`ucv`
User control variable string.

`lcv`
Loop control variable string.

`sigIdx`
Integer offset into block signal.

Returns

The input signal label based on the type of input signal, i.e., `U`, `X`, `B`, or `G`.

Description

`LibBlockInputSignal` returns the input signal label based on the type of input signal, i.e., `U`, `X`, `B`, or `G`. For example, `U.Vin[2]` is generated for a wide input signal named `Vin` (nonloop rolling case). In general, `LibBlockInputSignal` returns:

`rtGROUND`	If input signal is `GROUND`.
`Source.block[ucv]`	If `ucv` is specified.
`u%<portNum>[lcv]`	If `lcv` specified and signal is wide.
`Source.block`	If `lcv` is specified and signal is scalar.
`Source.block[sigIdx]`	Otherwise, where `[sigIdx]` is optional for wide signals.

where `Source` is `U`, `X`, or `B`.

LibBlockIWork

Purpose

Determines the appropriate `iwork` element

Syntax

```
%<LibBlockIWork(iwork, ucv, lcv, idx)>
```

Arguments

`iwork`
Reference to `iwork` identifier or `""` if there are no `IWorkDefine` records in the block.

`ucv`
User control variable string.

`lcv`
Loop control variable string.

`idx`
Integer index into this `iwork`.

Returns

The appropriate `iwork` element.

Description

This function returns the appropriate `iwork` element. In order to roll a block's `iwork`, it must first be defined with `LibDefineIWork`.

Purpose

Determines the address of a block's matrix parameter

Syntax

```
%<LibBlockMatrixParameterAddr(param, rucv, rlcv, ridx, cucv, …
   clcv, cidx)>
```

Arguments

`param`
Reference to a block parameter identifier.

`rucv`
Row user control variable string.

`rlcv`
Row loop control variable string (*Not Supported*).

`ridx`
Integer row index.

`cucv`
Column user control variable string.

`clcv`
Column loop control variable string (*Not Supported*).

`cidx`
Integer column index.

Returns

The address of a block's matrix parameter.

Description

`LibBlockMatrixParameterAddr` determines the address of a block's matrix parameter. Loop rolling is currently not supported, and generates an error if requested (i.e., if `rlcv` or `clcv` is not null). This also produces an error if the parameter passed is not of type `Matrix`.

LibBlockMatrixParameter

Purpose

Determines the appropriate matrix parameter for a block given the row and column user control variable, loop control variable, and index.

Syntax

```
%<LibBlockMatrixParameter(param, rucv, rlcv, ridx, cucv, …
   clcv, cidx)>
```

Arguments

`param`
Reference to a block parameter identifier.

`rucv`
Row user control variable string.

`rlcv`
Row loop control variable string (*Not Supported*).

`ridx`
Integer row index.

`cucv`
Column user control variable string.

`clcv`
Column loop control variable string (*Not Supported*).

`cidx`
Integer column index.

Returns

A reference to a block's matrix parameter.

Description

`LibBlockMatrixParameter` determines the appropriate matrix parameter for a block given the row and column user control variable, loop control variable, and index. Loop rolling is currently not supported, and will generate an error if requested (i.e., if `rlcv` or `clcv` is not null). This also produces an error if the parameter passed is not of type `Matrix`.

Purpose Determines the appropriate block mode

Syntax

```
%<LibBlockMode(ucv, lcv, modeIdx)>
```

Arguments

`ucv`
User control variable string.

`lcv`
Loop control variable string.

`modeIdx`
Integer mode index.

Returns The appropriate block mode based on `ucv`, `lcv`, and `modeIdx`.

Description `LibBlockMode` returns the appropriate block mode.

LibBlockOutportLocation

Purpose

Determines the appropriate identifier for an `outport` block

Syntax

```
%<LibBlockOutportLocation(ucv, lcv, sigIdx)>
```

Arguments

`ucv`
User control variable string.

`lcv`
Loop control variable string.

`sigIdx`
Integer offset into block signal.

Returns

The appropriate identifier for an `outport` block.

Description

`LibBlockOutportLocation` returns the appropriate identifier for an outport block.

Output location is `Y`:

`Y.block[ucv]`	`ucv` is specified.
`y0[lcv]`	`lcv` is specified and signal is wide.
`Y.block`	`lcv` is specified and signal is scalar.
`Y.block[sigIdx]`	Otherwise.

Output location is `B`:

`B.block[ucv]`	`ucv` is specified.
`y0[lcv]`	`lcv` is specified and signal is wide.
`B.block`	`lcv` is specified and signal is scalar.
`B.block[sigIdx]`	Otherwise.

Notes: The index is appropriately replaced with `ucv` or `lcv` when specified (`ucv` has higher precedence than `lcv`).

The width of the output port is determined by the width of the input port.

LibBlockOutputSignal

Purpose

Determines the block's output signal.

Syntax

```
%<LibBlockOutputSignal(portIdx, ucv, lcv, sigIdx)>
```

Arguments

`portNum`
Integer port number.

`ucv`
User control variable string.

`lcv`
Loop control variable string.

`sigIdx`
Integer offset into block signal.

Returns

The block's output signal.

Description

`LibBlockOutputSignal` returns the block's output signal. The result is determined by the values of `ucv`, `lcv`, and `sigIdx`. The result is as follows:

`B.block[ucv]`	If `ucv` is specified.
`y%<portIdx>[lcv]`	If `lcv` is specified and signal is wide.
`B.block`	If `lcv` is specified and signal is scalar.
`B.block[sigIdx]`	Otherwise, where `[sigIdx]` is optional for wide signals.

Notes: The precedence is `ucv`, `lcv`, then `sigIdx`. That is, if `ucv` and `lcv` are both specified, `ucv` takes precedence over `lcv`. Also, the vector index is only added for wide signals.

Purpose

Determines a block's parameter in the appropriate form depending on the state of loop rolling

Syntax

```
%<LibBlockParameter(param, ucv, lcv, sigIdx)>
```

Arguments

`param`
Reference to a block parameter identifier.

`ucv`
User control variable string.

`lcv`
Loop control variable string.

`sigIdx`
Integer offset into signal.

Returns

A block's parameter in the appropriate form depending on the state of loop rolling, `InlineParameters`, and the specified index.

Description

`LibBlockParameter` returns a block's parameter in the appropriate form depending on the state of loop rolling, `InlineParameters`, and the specified index. The user control variable (`ucv`) has higher precedence than `lcv` and `sigIdx`. That is, the following results if `ucv` is specified:

```
%<LibBlockParameter(Gain, "ucv", "", 0)>
```

`rtP.block.prm`	If `ucv` is specified and `prm` is scalar.
`rtP.block.prm[ucv]`	If `ucv` is specified and `prm` is vector.

Otherwise, the `ucv` is specified as `""`, and the result depends on the truth table below. Note that loop rolling is true whenever the loop control variable (`lcv`) is not null.

Assume:

```
%<LibBlockParameter(Gain, "", "i", 0)>
P.blk.Gain[0] = 4.55
```

Case	Rolling	InlineParameters	Type	Result	P Needed in Memory
1	0	1	scalar	4.55	no
2	1	1	scalar	4.55	no
3	0	1	vector	4.55	no
4	1	1	vector	p_Gain[i]	yes
5	0	0	scalar	rtP.blk.Gain	no
6	1	0	scalar	rtP.blk.Gain	no
7	0	0	vector	rtP.blk.prm[0]	no
8	1	0	vector	p_Gain[i]	yes

Note: Case 4 maintains the parameter even though `InlineParameters` is selected.

Do not use this function if you're using the result to get the address of a parameter. The reason is that when you're inlining parameters you'll end up referencing a number (i.e., `&4.55`). To avoid this situation use library function

```
%<LibBlockParameterAddr(param, ucv, lcv, sigIdx)>
```

Example

Assuming `Gain` is the second block parameter, these are equivalent

```
%assign param = LibBlockParameter(Gain, "", "", 0)
%assign param = LibBlockParameter(Parameter[1], "", "", 0)
```

This routine does not work for matrix parameters. Use `LibBlockMatrixParamter` when accessing a block's matrix parameter. If a matrix parameter is accessed via this routine, the reported error message is:

```
%exit %<Type> block %<Name> must access %<param.Name> via…
LibBlockMatrixParameter.
```

Purpose Determines the address of a block parameter.

Syntax `%<LibBlockParameterAddr(param, ucv, lcv, idx)>`

Arguments

`param`
Reference to a block parameter identifier.

`ucv`
User control variable string.

`lcv`
Loop control variable string.

`idx`
Integer index.

Returns The address of a block parameter.

Description `LibBlockParameterAddr` returns the address of a block parameter as:

`&P.block.param[ucv]`	If `ucv` is specified.
`&P.block.param[lcv]`	If `lcv` is specified.
`&P.block.param[idx]`	Otherwise.

This routine does not work for matrix parameters. Use `LibBlockMatrixParamterAddr` when accessing a block's matrix parameter.

LibBlockPWork

Purpose Determines the appropriate pwork element

Syntax

```
%<LibBlockPWork(pwork, ucv, lcv, idx)>
```

Arguments

pwork
Reference to pwork identifier or "" if there are no PWorkDefine records in the block.

ucv
User control variable string.

lcv
Loop control variable string.

idx
Integer index into this pwork.

Returns The appropriate pwork element.

Description LibBlockPWork returns appropriate pwork element.

Purpose Determines the appropriate rwork element

Syntax

```
%<LibBlockRWork(rwork, ucv, lcv, idx)>
```

Arguments

rwork
Reference to rwork identifier or "" if there are no RWorkDefine records in the block.

ucv
User control variable string.

lcv
Loop control variable string.

idx
Integer index into this rwork.

Returns The appropriate rwork element.

Description LibBlockRWork returns appropriate rwork element.

LibBlockSrcSignalIsDiscrete

Purpose

Determines if the driving signal is discrete

Syntax

```
%<LibBlockSrcSignalIsDiscrete(portNum, sigIdx)>
```

Arguments

`portNum`
Integer data input port number.

`sigIdx`
Integer offset into block input signal.

Returns

1 (yes) or 0 (no).

Description

`LibBlockSrcSignalIsDiscrete` determines if the driving signal is discrete.

Purpose

Call this function from inside `BlockTypeSetup` to cache a `#define` statement.

Syntax

```
%<LibCacheDefine(buffer)>
```

Arguments

`buffer`
Buffer of `#define` statements to be cached.

Description

`LibCacheDefine` should be called from inside `BlockTypeSetup` to cache a `#define` statement. Each call to this function appends your buffer to the existing cache buffer. The `#define` statements are placed inside *`model`*`.h`.

Example

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

LibCacheFunctionPrototype

Purpose

Call this function from inside `BlockTypeSetup` to cache a function prototype.

Syntax

```
%<LibCacheFunctionPrototype(buffer)>
```

Arguments

`buffer`
Buffer of function prototypes to be cached.

Description

This function 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)>
```

Purpose

Call this function from inside `BlockInstanceSetup` to cache global block parameter data.

Syntax

```
%<LibCacheGlobalPrmData(buffer)>
```

Arguments

`buffer`
Buffer of global data.

Description

This function should be called from inside `BlockInstanceSetup` to cache global block parameter data. Each call to this function appends your buffer to the existing cache buffer. The global data is placed inside `model.prm`.

Example

```
%openfile buffer
   real_T A[][] = {
       { 1.0, 0.0, 0.0 },  /* row 1 */
       { 1.0, 2.0, 0.0 },  /* row 2 */
       { 1.0, 0.0, 3.0 }   /* row 3 */
   }
%closefile buffer
%<LibCacheGlobalPrmData(buffer)>
```

See Also

`LibCacheNonFiniteAssignment`

LibCacheInclude

Purpose

Call this function from inside `BlockTypeSetup` to cache a `#include` statement.

Syntax

```
%<LibCacheInclude(buffer)>
```

Arguments

`buffer`
Buffer of `#include` statements to be cached.

Description

This function should be called from inside `BlockTypeSetup` to cache a `#include` statement. 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 "mystuff.h"
%closefile buffer
%<LibCacheInclude(buffer)>
```

Purpose

Call this function from inside `BlockInstanceSetup` to cache assignments that need to be placed in the registration function because of nonfinite initialization.

Syntax

```
%<LibCacheNonFiniteAssignment(buffer)>
```

Arguments

`buffer`
Buffer to be cached for placement inside the model's registration function.

Description

This function should be called from inside `BlockInstanceSetup` to cache assignments that need to be placed in the registration function because of nonfinite initialization. That is, the `rtInfs`, `rtNaNs`, and `rtMinusInfs` parameters are initialized to zero until the registration function is called, re-initializing them to their appropriate value. Each call to this function appends your buffer to the existing cache buffer.

Example

```
%openfile buffer
   rtP.block.param_1 = rtInf;
   rtP.block.param_i = rtNaN;
   rtP.block.param_n = rtMinusInf;
%closefile buffer
%<LibCacheNonFiniteAssignment(buffer)>
```

LibContinuousState

Purpose

Determines the block continuous state with optional scalar expansion

Syntax

```
%<LibContinuousState(ucv, lcv, idx)>
```

Arguments

`ucv`
User control variable string.

`lcv`
Loop control variable string.

`idx`
Integer offset into block states.

Returns

The block continuous state with optional scalar expansion.

Description

`LibContinuousState` returns the block continuous state with optional scalar expansion.

`X.c.block[ucv]`	If `ucv` is specified.
`xc[lcv]`	If `lcv` is specified.
`X.c.block[idx]`	Otherwise.

Purpose

Determines the appropriate control port input signal depending on the source of the input signal

Syntax

```
%<LibControlPortInputSignal(portNum, sigIdx)>
```

Arguments

`portNum`
Integer control port number, starting from 0.

`sigIdx`
Integer offset into the signal, i.e., current index of `foreach`.

Returns

The appropriate control port input signal depending on the source of the input signal (i.e., `Ui`, `Xi`, `Bi`, or `Gi`).

Description

`LibControlPortInputSignal` returns the appropriate control port input signal depending on the source of the input signal.

LibConvertZCDirection

Purpose

Converts Real-Time Workshop zero-crossing direction to a SimStruct representation

Syntax

```
%<LibConvertZCDirection(direction)>
```

Arguments

`direction`
Zero-crossing direction identifier from the block `ZCEvent` record.

Description

`LibConvertZCDirection` converts RTW zero-crossing direction to a SimStruct representation.

Rising	`RISING_ZERO_CROSSING`
Any	`ANY_ZERO_CROSSING`
Falling	`FALLING_ZERO_CROSSING`

Purpose Determines the width of an input port

Syntax `%<LibDataInputPortWidth(portNum)>`

Arguments

`portNum`
Integer input port number (starting from 0).

Returns The width of an input port.

Description `LibDataInputPortWidth` returns the width of an input port.

LibDataOutputPortWidth

Purpose Determines the width of the block's output port

Syntax `%<LibDataOutputPortWidth(portNum)>`

Arguments `portNum`
Integer port number (starting from 0).

Returns The width of the blocks output port.

Description `LibDataOutputPortWidth` determines the width of the blocks output port.

Purpose Determines the appropriate data store memory value

Syntax

```
%<LibDataStoreMemory(ucv, lcv, variableIdx)>
```

Arguments

`ucv`
User control variable string.

`lcv`
Loop control variable string.

`variableIdx`
Integer index into the data store memory variable.

Returns The appropriate data store memory value.

Description `LibDataStoreMemory` determines the appropriate data store memory value.

LibDeclareRollVariables

Purpose

Declares the necessary local variables required for loop rolling

Syntax

```
%<LibDeclareRollVariables(block, sigOffset, numIterations,\
                          rolledCount, rollVars)>
```

Arguments

`block`
Reference to the block record.

`sigOffset`
Integer signal offset of current roll region.

`numIterations`
Integer number of iterations in current roll region.

`rolledCount`
Integer number of times the Target Language Compiler has called `roller` for a given `RollRegion`.

`rollVars`
String vector of variables to declare. Table 4-1 lists the valid `rollVars`.

Table 4-1: Valid rollVars

	Declare All	Declare Individual
Inputs	`U`	`ui`
Outputs	`Y`	`yi`
Continuous states	`Xc, Xc`	
Discrete states	`Xd, Xd`	
Parameters	`P`	`<param>/name`
Real-work	`RWork`	`<rwork>/name`
Integer-work	`IWork`	`<iwork>/name`
Pointer-work	`PWork`	`<pwork>/name`
Mode	`Mode`	

Table 4-1: Valid rollVars (Continued)

	Declare All	Declare Individual
Previous zero-crossing	`PZC`	
Data store memory	`DSM`	

For example, `rollVars = ["U", "<param>/Gain"]` declares all nonscalar block inputs and the specific parameter `"Gain"`. See `gain.tlc` for an example of this function.

Description

`LibDeclareRollVariables` declares the necessary local variables required for loop rolling.

See Also

`roller.tlc`

LibDefineIWork

Purpose

Call this function from inside the block's `BlockInstanceSetup` function to define the specified `iwork` in the block.

Syntax

```
%<LibDefineIWork(block, name, width)>
```

Arguments

`block`
Reference to the block record.

`name`
String that you want to call the `iwork`.

`width`
Integer width of the `iwork`.

Description

This call should be made from inside the block's `BlockInstanceSetup` function, and adds the specified `iwork` to the block. The function creates and maintains an internal record for the `iwork` definition. For example, a block may have `iwork` records for system enable.

```
%<LibDefineIWork(block, "SystemEnable", 1)>
%<LibDefineIWork(block, "IcNeedsLoading", 1)>
```

Internally this creates a block record

```
NumIWorkDefines    2
IWorkDefine {
   Name            "SystemEnable"
   Width           1
}
IWorkDefine {
   Name            "IcNeedsLoading"
   Width           1
}
SystemEnable       IWorkDefine[0]
ICNeeedsLoading    IWorkDefine[1]
```

Note that `SystemEnable` and `IcNeedsLoading` are references to `IWorkDefine[0]` and `IWorkDefine[1]`, respectively, and are added by the system file, which executes the block's `BlockInstanceSetup` function.

Purpose

Call this function from inside the block's `BlockInstanceSetup` function to define the specified `pwork` in the block.

Syntax

```
%<LibDefinePWork(block, name, width)>
```

Arguments

`block`
Reference to the block record.

`name`
String that you want to call the `pwork`.

`width`
Integer width of the `pwork`.

Description

This call should be made from inside the block's `BlockInstanceSetup` function, and adds the specified `pwork` to the block. The function creates and maintains an internal record for the `pwork` definition. For example, a block may have a `pwork` record for data logging.

```
%<LibDefinePWork(block, "LoggedData", 3)>
```

Internally this creates a block record

```
NumPWorkDefines     1
PWorkDefine {
   Name             "LoggedData"
   Width            3
}
LoggedDat           PWorkDefine[0]
```

Note that `LoggedData` is a reference to `PWorkDefine[0]`, which is added by the system file which executes the block's `BlockInstanceSetup` function.

LibDefineRWork

Purpose

Call this function from inside the block's `BlockInstanceSetup` function to define the specified `rwork` definition in the block.

Syntax

```
%<LibDefineRWork(block, name, width)>
```

Arguments

`block`
Reference to the block record.

`name`
String that you want to call the `rwork`.

`width`
Integer width of this `rwork`.

Description

This call should be made from inside the block's `BlockInstanceSetup` function, and adds the specified `rwork` definition to the block. The function creates and maintains an internal record for the `rwork` definition, removing the Simulink definition if necessary.

```
%<LibDefineRWork(block, "PrevT", 1)>
%<LibDefineRWork(block, "PrevU", 3)>
```

Internally this creates a block record

```
NumRWorkDefines    2
RWorkDefine {
   Name            "PrevT"
   Width           1
}
RWorkDefine {
   Name            "PrevU"
   Width           3
}
PrevT              RWorkDefine[0]
PrevU              RWorkDefine[1]
```

Note that `PrevT` and `PrevU` are references to `RWorkDefine[0]` and `RWorkDefine[1]`, respectively, and are added by the system file, which executes the block's `BlockInstanceSetup` function.

Purpose

Determines a block's discrete state with optional scalar expansion

Syntax

```
%<LibDiscreteState(ucv, lcv, idx)>
```

Arguments

`ucv`
User control variable string.

`lcv`
Loop control variable string.

`idx`
Integer offset into block states.

Returns

Block's discrete state with optional scalar expansion.

Description

`LibDiscreteState` return a block's discrete state with optional scalar expansion.

`X.d.block[ucv]`	If `ucv` is specified.
`xd[lcv]`	If `lcv` is specified.
`X.d.block[idx]`	Otherwise.

LibExternalResetSignal

Purpose

Determines the appropriate reset signal into the reset port depending on the source of input signal

Syntax

```
%<LibExternalResetSignal(portNum, sigIdx)>
```

Arguments

`portNum`
Integer reset port number, starting from 0.

`sigIdx`
Integer offset into the signal, i.e., current index of `foreach`.

Returns

The appropriate reset signal into the reset port depending on the source of input signal.

Description

`LibExternalResetSignal` returns the appropriate reset signal into the reset port depending on the source of input signal (i.e., `Ui`, `Xi`, `Bi`, or `Gi`).

Purpose

Places code at the top or bottom of the model's header file

Syntax

```
%<LibHeaderFileCustomCode(buffer, location)>
```

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
`"header"` (place at top of file).
`"trailer"` (place at bottom of file).

Description

Use this function to place code at the top or bottom of the model's header file by specifying `"header"` or `"trailer"`, respectively. Each call to this function appends your buffer to the internal cache buffer.

LibIndexStruct

Purpose

Returns an optional index for a structure field

Syntax

```
%<LibIndexStruct(width, ucv, lcv, idx)>
```

Arguments

`width`
Integer width of variable.

`ucv`
User control variable string.

`lcv`
Loop control variable string.

`idx`
Integer index.

Returns

An optional index for a structure field.

Description

`LibIndexStruct` returns an optional index for a structure field. An index into to the signal is returned for wide signals. Nothing is returned for scalar signals (this scalar expands them).

`""`	Signal is scalar (width == 1).
`[ucv]`	Width > 1, `ucv` is specified.
`[lcv]`	Width > 1, `lcv` is specified.
`[idx]`	Otherwise.

Purpose Determines if a block is discrete based on its task identifier (TID)

Syntax

```
%<LibIsDiscrete(tid)>
```

Arguments

`tid`
Task identifier (i.e., integer index into `SampleTime`).

Returns 1 if discrete, otherwise returns 0.

Description Determines if a block is discrete based on its TID. Note that TIDs equal to `triggered` or `constant` are not discrete.

LibIsEmpty

Purpose

Determines if input is an empty string, an empty vector, or an empty matrix array

Syntax

```
%<LibIsEmpty(input)>
```

Arguments

`input`
Input variable.

Returns

1 if input is an empty string: `""`, an empty vector: `[]`, or an empty matrix array: `[[] []]`.

Description

`LibIsEmpty` returns 1 if input is an empty string: `""`, or an empty vector: `[]`, or an empty matrix array: `[[] []]`.

Purpose Determines if `expr1` equals `expr2`

Syntax `%<LibIsEqual(expr1, expr2)>`

Arguments

`expr1`
First expression.

`expr2`
Second expression.

Returns 1 if `expr1` equals `expr2`, otherwise 0 is returned.

Description `LibIsEqual` returns 1 if `expr1` equals `expr2`, otherwise it returns 0. Note that different type expressions always return 0. That is, “0” does not equal 0.

LibIsFinite

Purpose Determines if the number is finite

Syntax `%<LibIsFinite(value)>`

Arguments `value`
Any number including `rtInf`, `rtMinusInf`, and `rtNaN`.

Returns 1 if the number is finite, otherwise, it returns 0.

Description `LibIsFinite` returns 1 if the number is finite, otherwise, it returns 0.

Purpose

Determines the appropriate source signal given the mapping source and mapping index

Syntax

```
%<LibMapSignalSource(mappingSource, mappingIndex, ucv, lcv)>
```

Arguments

`mappingSource`
String `"U"`, `"X"`, `"B"`, or `"G"`.

`mappingIndex`
Integer index into the map for `U`, `X`, `B`, or `G`.

`ucv`
User control variable string.

`lcv`
Loop control variable string.

Returns

The appropriate source signal given the mapping source and mapping index.

Description

`LibMapSignalSource` returns the appropriate source signal given the mapping source and mapping index. Valid mapping sources are `U`, `X`, `B`, and `G`. The mapping index is the index into these maps.

Example

A wide input signal named `Vin` may produce the following result for `LibMapSignalSource(U, mappingIndex, "", "")`

```
U.Vin[2]
```

LibMaxBlockIOWidth

Purpose Determines the maximum width of the input or output ports

Syntax `%<LibMaxBlockIOWidth()>`

Arguments none

Returns The maximum width of the output or input ports.

Description If the block has output ports, `LibMaxBlockIOWidth` returns the maximum width of the output ports, otherwise, it returns the maximum width of its input ports.

Purpose Determines the maximum width of all the input ports

Syntax `%<LibMaxDataInputPortWidth()>`

Arguments none

Returns The maximum width of all the input ports.

Description `LibMaxDataInputPortWidth` returns the maximum width of all the input ports.

LibMaxDataOutputPortWidth

Purpose Determines the maximum width of all the output ports.

Syntax `%<LibMaxDataOutputPortWidth()>`

Arguments none

Returns The maximum width of all the output ports.

Description `LibMaxDataOutputPortWidth` returns the maximum width of all the output ports.

Purpose

Places code at the top or bottom of the model's registration function

Syntax

```
%<LibMdlRegCustomCode(buffer, location)>
```

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
`"header"` (place at top of function).
`"trailer"` (place at bottom of function).

Description

Use this function to place code at the top or bottom of the model's registration function by specifying `"header"` or `"trailer"`, respectively. Each call to this function appends your buffer to the internal cache buffer.

LibMdlStartCustomCode

Purpose Places code at the top or bottom of the `MdlStart` function

Syntax

```
%<LibMdlStartCustomCode(buffer, location)>
```

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
`"header"` (place at top of function).
`"trailer"` (place at bottom of function).

Description Use this function to place code at the top or bottom of the `MdlStart` function by specifying `"header"` or `"trailer"`, respectively. Each call to this function appends your buffer to the internal cache buffer.

Purpose

Places code at the top or bottom of the `MdlTerminate` function

Syntax

```
%<LibMdlTerminateFcnCustomCode(buffer, location)>
```

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
`"header"` (place at top of function).
`"trailer"` (place at bottom of function).

Description

Use this function to place code at the top or bottom of the `MdlTerminate` function by specifying `"header"` or `"trailer"`, respectively. Each call to this function appends your buffer to the internal cache buffer.

LibOptionalMatrixWidth

Purpose

Determines the optional width of a matrix

Syntax

```
%<LibOptionalMatrixWidth(nRows, nCols)>
```

Arguments

`nRows`
Integer number of rows.

`nCols`
Integer number of columns.

Returns

Returned Value	Data Type
Nothing	Scalars
`[nRows]` or `[nCols]`	Row or column vector
`[nRows][nCols]`	Matrices

Description

`LibOptionalMatrixWidth` returns the optional width of a matrix.

Purpose Determines if length is greater than 1

Syntax `%<LibOptionalVectorWidth(length)>`

Arguments

`length`
Integer vector length.

Returns `[length]` if `length` is greater than 1, otherwise it returns an empty string.

Description `LibOptionalVectorWidth` returns `[length]` if `length` is greater than 1, otherwise it returns an empty string.

LibPathName

Purpose

Determines the full path of a system

Syntax

```
%<LibPathName(name)>
```

Arguments

`name`
String name of system (e.g., `"<S5>/foo_system"`).

Returns

The full path of a system.

Description

`LibPathName` returns the full path of a system. This is a recursive function. Note that the expanded name can be used to locate a Simulink block using `open_system` at the MATLAB prompt with the result of this function.

Purpose

Determines the appropriate element for previous zero-crossing state based on ucv, lcv, and pzcIdx.

Syntax

```
%<LibPrevZCState(ucv, lcv, pzcIdx)>
```

Arguments

ucv
User control variable string.

lcv
Loop control variable string.

pzcIdx
Integer previous zero-crossing index.

Returns

The appropriate element for the previous zero-crossing state based on ucv, lcv, and pzcIdx.

Description

LibPrevZCState returns the appropriate element for previous zero-crossing state based on ucv, lcv, and pzcIdx.

LibPrmFileCustomCode

Purpose

Places code at the top or bottom of the model's parameter file

Syntax

```
%<LibPrmFileCustomCode(buffer, location)>
```

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
`"header"` (place at top of file).
`"trailer"` (place at bottom of file).

Description

Use this function to place code at the top or bottom of the model's parameter file by specifying `"header"` or `"trailer"`, respectively. Each call to this function appends your buffer to the internal cache buffer.

Purpose Places code at the top or bottom of the model's registration file

Syntax

```
%<LibRegFileCustomCode(buffer, location)>
```

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
`"header"` (place at top of file).
`"trailer"` (place at bottom of file).

Description Use this function to place code at the top or bottom of the model's registration file by specifying `"header"` or `"trailer"`, respectively. Each call to this function appends your buffer to the internal cache buffer.

LibRenameParameter

Purpose

Renames the parameter and creates a new reference to the parameter

Syntax

```
%<LibRenameParameter(block, param, newName)>
```

Arguments

`block`
Reference to the block record.

`param`
Reference to the block parameter identifier.

`newName`
New string name for the parameter.

Description

This call should be made from inside the block's `BlockInstanceSetup` function. This function

- Renames the parameter to the name specified.
- Creates a new reference to the parameter by that name.

Purpose

Places code at the top of the model's source file

Syntax

```
%<LibSourceFileCustomCode(buffer, location)>
```

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
`"header"` (place at top of file).

Description

Use this function to place code at the top of the model's source file by specifying `"header"`. Custom code is not allowed at the bottom of `model.c` since placing it at the top of `model.reg` has the same effect (`model.c` includes `model.reg` as its last statement).

Be careful placing code in `model.c` if file splitting is an issue. When code is needed in each split file, place it in `model.h` instead of `model.c`.

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

LibSystemDerivativeCustomCode

Purpose

Places code at the top or bottom of a system's derivative function

Syntax

```
%<LibSystemDerivativeCustomCode(buffer, location)>
```

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
`"header"` (place at top of function).
`"trailer"` (place at bottom of function).

Description

Use this function to place code at the top or bottom of a system's derivative function by specifying `"header"` or `"trailer"`, respectively.

This function generates an error if you attempt to add code to a subsystem that does not have any continuous states. Each call to this function appends your buffer to the internal cache buffer.

Purpose

Places code at the top or bottom of a system's disable function

Syntax

```
%<LibSystemDisableCustomCode(buffer, location)>
```

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
`"header"` (place at top of function).
`"trailer"` (place at bottom of function).

Description

Use this function to place code at the top or bottom of a system's disable function by specifying `"header"` or `"trailer"`, respectively.

This function generates an error if you attempt to add code to a subsystem that does not have a disable function. Each call to this function appends your buffer to the internal cache buffer.

LibSystemEnableCustomCode

Purpose

Places code at the top or bottom of a system's enable function

Syntax

```
%<LibSystemEnableCustomCode(buffer, location)>
```

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
`"header"` (place at top of function).
`"trailer"` (place at bottom of function).

Description

Use this function to place code at the top or bottom of a system's enable function by specifying `"header"` or `"trailer"`, respectively.

This function generates an error if you attempt to add code to a subsystem that does not have an enable function. Each call to this function appends your buffer to the internal cache buffer.

Purpose

Places code at the top or bottom of a system's initialize function

Syntax

```
%<LibSystemInitializeCustomCode(buffer, location)>
```

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
`"header"` (place at top of function).
`"trailer"` (place at bottom of function).

Description

Use this function to place code at the top or bottom of a system's initialize function by specifying `"header"` or `"trailer"`, respectively. Note that enable systems that are not configured to reset on enable, get inlined into `MdlStart`. For this case, the system's custom code is found in `MdlStart` above and below the enable system's initialization code.

Attempting to add initialization code to the root system will generate an error. For this case, use library function `LibMdlStartFcnCustomCode`.

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

LibSystemOutputCustomCode

Purpose Places code at the top or bottom of a system's output function

Syntax

```
%<LibSystemOutputCustomCode(buffer, location)>
```

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
`"header"` (place at top of function).
`"trailer"` (place at bottom of function).

Description Use this function to place code at the top or bottom of a system's output function by specifying `"header"` or `"trailer"`, respectively. Each call to this function appends your buffer to the internal cache buffer.

Purpose Places code at the top or bottom of a system's update function

Syntax

```
%<LibSystemUpdateCustomCode(buffer, location)>
```

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
`"header"` (place at top of function).
`"trailer"` (place at bottom of function).

Description Use this function to place code at the top or bottom of a system's update function by specifying `"header"` or `"trailer"`, respectively. Each call to this function appends your buffer to the internal cache buffer.

A

model.rtw

This appendix describes the contents of the *model*.rtw file, which is created from your block diagram during the Real-Time Workshop build procedure, and is for use with the Target Language Compiler. The contents of the *model*.rtw file is a "compiled" version of your block diagram. This appendix is provided so that you can modify the existing code generation or even create a new "code generator" to suit your needs. The general format of the *model*.rtw file is:

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

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. Changes will be documented with each release.

Table A-1: Model.rtw File Contents

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.
`GeneratedOn`	Date and time when the *model*.rtw file was generated.
`Solver`	Name of solver as entered in the **Simulink Parameters** dialog box.
`SolverType`	`FixedStep` or `VariableStep`.
`StartTime`	Simulation start time as entered in the **Simulink Parameters** dialog box.
`StopTime`	Simulation stop time.
`FixedStepOpts {`	Only written if `SolverType` is `FixedStep`.
`  FixedStep`	Step size to be used.
`}`	
`VariableStepOpts {`	Only written if `SolverType` is `VariableStep`.

Table A-1: Model.rtw File Contents (Continued)

Variable/Record Name	Description
RelTol	Relative tolerance.
AbsTol	Absolute tolerance.
Refine	Refine factor.
MaxStep	Maximum step size.
InitialStep	Initial step size.
MaxOrder	Maximum order for ode15s.
}	
DataLoggingOpts {	Data logging record describing the settings of Simulink simulation. (Parameters, workspace, I/O settings)
LogT	Name of Time variable or "" if not selected to be saved.
LogX	Name of State variable or "" if not selected to be saved.
LogY	Name of output variable, variable list "output1,output2", or "" if not selected to be saved.
LogYNCols	Number of columns in output variable. Not written if LogY == "".
LogXFinal	Name of final state variable or "" if not selected to be saved.
MaxRows	Maximum number of rows or 0 for no limit.
Decimation	Data logging interval.
}	
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.

Table A-1: Model.rtw File Contents (Continued)

Variable/Record Name	Description
`DirectFeedthrough`	Does model require its inputs in the `MdlOutput` function (yes/no)?
`NumContStates`	Total number of continuous states in the model. Continuous states appear in your model when you use continuous components (i.e., an Integrator block) that have state(s) that must be integrated by a solver such as `ode45`.
`NumDiscStates`	Total number of discrete states in the model. Discrete states appear in your model when you use discrete components (i.e., a Unit Delay block) that have state(s). The model state vector, `X`, is of length `NumContStates` plus `NumDiscStates` and contains the continuous states followed by the discrete states.
`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.
`NumNonsampledZCs`	Length of the model nonsampled zero-crossing vectors, one for the zero-crossing signals (`nonsampledZCs`) and one for 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`).
`NumRWork`	Length of the model real-work vector (`rwork`). Real-work elements are used by blocks that need to keep track of “real” variables between simulation steps. An example of a block that uses real-work elements is the Discrete Sine Wave block, which has discrete coefficients that are needed across simulation steps.

Table A-1: Model.rtw File Contents (Continued)

Variable/Record Name	Description
`NumIWork`	Length of the model integer-work vector (`iwork`). Integer-work elements are used by blocks that need to keep track of "integer" variables between simulation steps. An example of a block that uses integer-work elements is the Discrete Time Integrator block configured with an external initial condition source. The integer-work element is used as a Boolean to determine when to load the initial condition.
`NumPWork`	Length of the model pointer-work vector (`pwork`). Pointer-work elements are used by blocks that need to keep track of "pointer" variables between simulation steps. An example of a block that uses pointer-work elements is the To Workspace block, which uses a pointer-work element to keep track of logged data.
`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.
`InvariantConstants`	`yes` if invariant constants (i.e., inline-parameters) is "on", `no` if invariant constants is off.
`FundamentalStepSize`	Fundamental step size or 0.0 if one cannot be determined. Variable step solvers may have a fundamental step size of 0.0.
`NumSampleTimes`	Number of sample times in the model followed by `SampleTime` info records, giving the `TID` (task ID), an index into the sample time table, and the period and offset for the sample time.
`SampleTime {`	One record for each sample time.
`  TID`	Task ID for this sample time.

Table A-1: Model.rtw File Contents (Continued)

Variable/Record Name	Description
PeriodAndOffset	Period and offset for this sample time.
}	
RootSignals {	Signal and block information in the root "window."
NumSignals	Number of block output signals (including virtual) blocks.
Signal {	One record for each signal.
Block	[sysIdx, blockIdx] or block name string if a virtual block.
SigLabel	Signal label if present.
OutputPort	[outputPortIndex, outputPortWidth].
SignalSrc	Vector of length outputPortWith giving the location of the signal source.
}	
NumBlocks	Number of nonvirtual blocks in the root window of your model.
BlockSysIdx	System index for blocks in this subsystem.
BlockMap	Vector of length NumBlocks giving the blockIdx for each nonvirtual block in the subsystem.
}	
NumVirtualSubsystems	Total number of virtual subsystems in the model.
NumNonvirtalSubsystems	Total number of nonvirtual subsystems in the model.
Subsystem {	One record for each subsystem.
SysId	System identifier. Each subsystem in the model is given a unique identifier of the form S# (e.g., S3).
Name	Block name preceded with a <root> or <S#> token. The ID/Name values define an associative pair giving a mapping to the block full path block.

Table A-1: Model.rtw File Contents (Continued)

Variable/Record Name	Description
SLName	Unmodified Simulink name. This is only written if it is *not* equal to Name.
Virtual	Whether or not the subsystem is virtual.
NumSignals	Number of block output signals (including virtual) blocks.
Signal {	One record for each signal.
Block	[sysIdx, blockIdx] or block name string if a virtual block.
OutportBlock	This is only present 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].
SignalSrc	Vector of length outputPortWith giving the location of the signal source.
}	
NumBlocks	Number of nonvirtual blocks in the subsystem.
BlockSysIdx	System index for blocks in this subsystem.
BlockMap	Vector of length NumBlocks giving the blockIdx for each nonvirtual block in the subsystem.
}	
DataStores {	List of data stores in the block diagram.
NumDataStores	Number of named data stores.
DataStore {	One for each data store.
Name	Name of block declaring the data store.

Table A-1: Model.rtw File Contents (Continued)

Variable/Record Name	Description
SLName	Unmodified Simulink name. This is only written if it is *not* equal to Name.
MemoryName	Name of the data store memory region.
Identifier	Unique identifier across all data stores.
Index	[dataStoreIndex, dataStoreWidth].
InitValue	Initial value for the data store.
}	
}	
ExternalInputs {	External inputs to the block diagram.
NumExternalInputs	Number of external input records that follow.
ExternalInput {	One for each external input signal.
Identifer	Unique name across all external inputs.
SigIdx	[externalInputVectorIndex, signalWidth].
SigLabel	Signal label entered by user.
}	
}	
ExternalInputsMap	Matrix of dimension (NumModelInputs,2), which gives a mapping from external input vector index (Ui) into the ExternalInputs structure: [externalInputsIndex, signalOffset]. Only written if NumModelInputs > 0.
BlockOutputs {	List of block output signals in the block diagram.
NumBlockOutputs	Number of data output port signals.
BlockOutput {	One for each data output signal.

Table A-1: Model.rtw File Contents (Continued)

Variable/Record Name	Description
Identifier	Unique variable name across all block outputs.
SigSrc	[systemIndex, blockIndex, outputPortIndex].
SigIdx	[blockIOVectorIndex, signalWidth].
SigConnected	Vector of length signalWith where each element is either a 1 or 0 indicating whether or not the corresponding output signal is connected.
SigLabel	Signal label entered by user.
TestPoint	yes/no. Has this signal been marked as a test point in the block diagram?
Required	yes/no. Is this signal required to be in the block I/O structure (i.e., the signal is needed across function boundaries)?
}	
}	
BlockOutputsMap	Matrix of dimension (NumBlockSignals,2), which gives a mapping from a block I/O vector index (Bi) into the BlockOutputs structure: [blockOutputsIndex, signalOffset]. Only written if NumBlockSignals > 0.
StatesMap	Matrix of dimension (NumContStates,3), which gives a mapping from a continuous or discrete state vector index (Xi) to a block: [systemIndex, blockIndex, stateOffset]. If stateOffset is less than ContStates[0], then Xi maps to the continuous state at stateOffset in the block, otherwise Xi maps to the discrete state at stateOffset–ContStates[0] in the block. Only written if NumContStates+NumDiscStates > 0.
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.
IllegalMTaskTrans	no

Table A-1: Model.rtw File Contents (Continued)

Variable/Record Name	Description
InMask	no
AlgebraicLoopId	0
ContStates	[0,0]
DiscStates	[0,0]
RWork	[0,0]
IWork	[0,0]
PWork	[0,0]
ModeVector	[0,0]
NonsampledZCs	[0,0]
ZCEvents	[0,0]
RollRegions	[]
NumDataInputPorts	0
NumControlPorts	0
NumDataOutputPorts	0
Parameters	[0,0,0]
NumRWorkDefines	0
NumIWorkDefines	0
NumPWorkDefines	0
}	

In general, a model can consist of multiple systems. There is one system for the root and one for each nonvirtual subsystem. Each descendent system of the root system is written out using Pascal ordering (deepest first) to avoid forward references. Within each system is a sorted list of blocks.

Table A-2: Model.rtw File Contents — System Record

Variable/Record Name	Description
`System {`	One for each system in the model.
`Type`	`root`, `enable`, `trigger`, `enable_with_trigger`, or `function-call`.
`Tag`	Only written if block has a non-empty Simuilnk "tag" property.
`Name`	Name of system.
`SLName`	Unmodified Simulink name. This is only written if it is *not* equal to `Name`.
`Identifier`	Unique identifier across all blocks.
`SubsystemBlockIdx`	`[systemIndex, blockIndex]`. Not present if `Type` is root.
`InitializeFcn`	Name of initialize function for enable systems that are configured to reset states.
`OutputFcn`	Name of output function for `enable` systems.
`UpdateFcn`	Name of update function for `enable` systems.
`DerivativeFcn`	Name of derivative function for `enable` systems that have continuous states.
`EnableFcn`	Name of disable function for `enable` or `enable_with_trigger` systems.
`DisableFcn`	Name of disable function for `enable` or `enable_with_trigger` systems.
`ZeroCrossFcn`	Name of nonsampled zero-crossing function for `enable` systems using variable step solver.

Table A-2: Model.rtw File Contents — System Record (Continued)

Variable/Record Name	Description
`OutputUpdateFcn`	Name of output/update function for `trigger` or `enable_with_trigger` systems.
`NumBlocks`	Number of blocks in the system.
`Block {`	One for each block in the system.
`Type`	Block type, i.e., 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 reside in a masked subsystem.
`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.
`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`).

Table A-2: Model.rtw File Contents — System Record (Continued)

Variable/Record Name	Description
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.
IllegalMTaskTrans	yes if there is a sample time transition that would cause multitasking problems. no if transition is fine. "No-RatesEqual" if there is a continuous to discrete or discrete to continuous, but the rates are equal.
AlgebraicLoopId	This ID identifies the loop this block is in. If this field is not present, the ID is 0 and the block is not part of an algebraic loop.
ContStates	Specified as [N,I] where N is number of continuous states and I is the index into the state vector, X. Not present if N==0.
DiscStates	Specified as [N,I] where N is number of discrete states and I is the index into the state vector, X. Not present if N==0.
RWork	Specified as [N,I] where N is the number of real-work elements and I is the index into rwork. Not present if N==0.
IWork	Specified as [N,I] where N is the number of integer-work elements and I is the index into iwork. Not present if N==0.
PWork	Specified as [N,I] where N is the number of pointer-work elements and I is the index into pwork. Not present if N==0.
ModeVector	Specified as [N,I] where N is the number of modes and I is the index into modeVect. Not present if N==0.
NonsampledZCs	Specified as [N,I], where N is the number of nonsampled zero-crossings and I is the index into the nonsampledZCs and nonsampledZCdirs vectors.
NonsampledZC {	One record for each nonsampled zero-crossing.
Index	Index of the block's zero-crossing.

Table A-2: Model.rtw File Contents — System Record (Continued)

Variable/Record Name	Description
Direction	Direction of zero-crossing: Falling, Any, Rising.
}	
ZCEvents	Specified as [N,I], where N is the number of zero-crossing events and I is the index into the zcEvents vector.
ZCEvent {	One record for each zero-crossing event.
Type	Type of zero-crossing: DiscontinuityAtZC, ContinuityAtZC, TriggeredDisconAtZC.
Direction	Direction of zero-crossing: Falling, Any, Rising.
}	
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.
RollRegions1	This is equivalent to RollRegions shifted left by 1. It is present for the blocks that collapse a vector to a scalar such as the vector Sum, Product, and MinMax blocks.
NumDataInputPorts	Number of data input ports. Only written if nonzero.
DataInputPort {	One record for each data input port.
Width	Length of the signal entering this input port.
SignalSrc	A vector of length Width where each element specifies the source signal. This is an index into the block I/O vector (Bi), an index into the state vector (Xi), an index into the external input vector (Ui), or unconnected ground (G0).
}	
NumControlPorts	Number of control (e.g., trigger or enable) input ports. Only written if nonzero.

Table A-2: Model.rtw File Contents — System Record (Continued)

Variable/Record Name	Description
`ControlPort {`	One record for control input port.
`Type`	Type of control port: `enable`, `trigger`, or `function-call`.
`Width`	Length of the signal entering this input port.
`SignalSrc`	A vector of length `Width` where each element specifies the source signal. This is an index into the block I/O vector (`Bi`), an index into the state vector (`Xi`), an index into the external input vector (`Ui`), or unconnected ground (`G0`).
`SignalSrcTID`	Vector of length `Width` giving the TID as an integer index, `trigger`, or `constant` identifier for each signal entering this control port.
`}`	
`NumDataOutputPorts`	Number of output ports. Only written if nonzero.
`DataOutputPortIndices`	Indices into `BlockOutputs` record. Only written if `NumDataOutputPorts > 0`.
`Parameters`	Specified as `[N,M,I]` where `N` is the number of `Parameter` records that follow, `M` is the number of modifiable parameter elements, and `I` is the starting index into the params vector. Not present if `N==0`.
`Parameter {`	One record for each parameter.
`Name`	Name of the parameter.
`Value`	Value of the parameter.
`String`	String entered in the Simulink block dialog box.

Table A-2: Model.rtw File Contents — 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.
}	
ParamSettings {	Optional record specific to block.
blockSpecificName	Block specific settings.
}	
}	
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.
}	

The following table describes the block specific records written for the Simulink blocks.

Table A-3: Model.rtw File Contents — Block Specific Records

Block Type: `AbsoluteValue`

No block specific records.

Block Type: `Backlash` (example with a backlash width of 2.08 and an initial output of [1.86, 2.38])

```
Parameter {
  Name                    "BacklashWidth"
  Value                   [2.08]
  String                  "2.08"
  StringType              "Expression"
}
 ParamSettings {
  InitialOutput           [1.86, 2.38]
}
NumRWorkDefines           1
RWorkDefine {             Used to store previous output and time values.
  Name                    PrevTY
  Width                   3
}
```

Block Type: `Clock`

No block specific records.

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

Block Type: `CombinatorialLogic` (example of 8-by-2 table):

```
Parameter {
  Name                    "TruthTable"
  Value                   Matrix(8,2)
[[0, 0]; [0, 1]; [0, 1]; [1, 0]; [0, 1]; [1, 0]; [1, 0]; [1,1];
  String                  "[0 0;0 1;0 1;1 0;0 1;1 0;1 0;1 1]"
  StringType              "Expression"
}
```

Block Type: `Constant` (example of a constant of 1:5):

```
Parameter {
  Name                    "Value"
  Value                   [1, 2, 3, 4, 5]
  String                  "1:5"
  StringType              "Expression"
}
```

Block Type: `DataStoreMemory`

Virtual. Not written to RTW file.

Block Type: `DataStoreRead`

```
ParamSettings {
  DataStore               Region index into data stores list.
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

```
}
```

Block Type: `DataStoreWrite`

```
ParamSettings {
  DataStore                Region index into data stores list.
}
```

Block Type: `Deadzone` (example of a deadzone block with a lower value of –3.3503 and an upper value of 1.4864).

```
Parameter {
  Name                     "LowerValue"
  Value                    [-3.3503]
  String                   "-3.3503"
  StringType               "Expression"
}
Parameter {
  Name                     "UpperValue"
  Value                    [1.4864]
  String                   "1.4864"
  StringType               "Expression"
}
```

Block Type: `Demux`

Virtual. Not written to *model*`.rtw` file.

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

Block Type: `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. This is 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. The following is an example of what will appear in the *model*`.rtw` file for an input of width 5.

```
NumRWorkDefines          4
RWorkDefine {
  Name                   TimeStampA
  Width                  1
}
RworkDefine {
  Name                   LastUAtTimeA
  Width                  5
}
RworkDefine {
  Name                   TimeStampB
  Width                  1
}
RworkDefine {
  Name                   LastUAtTimeB
  Width                  5
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

```
}
```

Block Type: `DigitalClock`

No block specific records.

Block Type: `DiscreteFilter`

See Model.rtw File Contents — Linear Block Specific Records on page A-51.

Block Type: `DiscreteIntegrator` (shown below is a limited integrator configured with an internal initial condition of `0`, an upper limit of `".75"`, and a lower limit of `"[-.25 0 -.75]"`).

```
Parameter {
  Name                  "InitialCondition"
  Value                 [0]
  String                "0"
  StringType            "Expression"
}
Parameter {
  Name                  "UpperSaturationLimit"
  Value                 [0.75]
  String                ".75"
  StringType            "Expression"
}
Parameter {
  Name                  "LowerSaturationLimit"
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

Value	[–0.25, 0, –0.75]
String	"[–.25 0 –.75]"
StringType	"Expression"
}	
NumRWorkDefines	0, 1, or 2.
RworkDefine {	Present if NumRWorkDefines is greater than 0.
Name	PrevT
Width	1
}	
RWorkDefine {	Present if NumRWorkDefines is 2.
Name	PrevU
Width	Equal to the width of the signal being integrated.
}	
ParamSettings {	
IntegratorMethod	ForwardEuler, BackwardEuler, or Trapezoidal
ExternalReset	none, rising, falling, or either
InitialConditionSource	internal or external
LimitOutput	on or off
ShowSaturationPort	on or off
ShowStatePort	on or off
ExternalX0	Only written when initial condition (IC) source is external. This is the initial value of the signal entering the IC port.
}	

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

BlockType: `DiscretePulseGenerator` (shown below is a discrete pulse generator with an amplitude of `1`, a period of `2` samples, a pulse width of `1` sample, and a phase delay of `0` samples).

```
 Parameter {
  Name                    "Amplitude"
  Value                   [1]
  String                  "1"
  StringType              "Expression"
}
Parameter {
  Name                    "Period"
  Value                   [2]
  String                  "2"
  StringType              "Expression"
}
Parameter {
  Name                    "PulseWidth"
  Value                   [1]
  String                  "1"
  StringType              "Expression"
}
ParamSettings {
  PhaseDelay              [0]
}
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

`NumIWorkDefines`	`1` (There is one integer work for each output signal.)
`IWorkDefine {`	
`Name`	`"ClockTicksCounter"`
`Width`	`1`
`}`	

Block Type: `DiscreteStateSpace`

See Model.rtw File Contents — Linear Block Specific Records on page A-51.

Block Type: `DiscreteTransferFcn`

See Model.rtw File Contents — Linear Block Specific Records on page A-51.

Block Type: `DiscreteZeroPole`

See Model.rtw File Contents — Linear Block Specific Records on page A-51.

Block Type: `Display`

No block specific records.

Block Type: `ElementaryMath`

`ParamSettings {`	
`Operator`	One of `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `sinh`, `cosh`, `tanh`, `exp`, `log`, `log10`, `floor`, `ceil`, `sqrt`, `reciprocal`, `pow`, or `hypot`
`}`	

Block Type: `EnablePort`

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

Virtual. Not written to *model*.rtw file.

Block Type: From

Virtual. Not written to *model*.rtw file.

Block Type: FromFile

```
ParamSettings {
  NumPoints                Number of data points.
  TUdata                   Data from the .mat file.
}
```

Block Type: FromWorkspace

```
ParamSettings {
  NumPoints                Number of data points.
  TUdata                   Data from the workspace variable(s).
}
```

Block Type: Fcn

The Fcn block is written out as an abstract syntax tree (AST). The following is an example for the expression "sin(u(1))+10".

```
ParamSettings {
  Expr                     "sin(u(1)) + 10"
}
ASTNode {
  Op                       "+"
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

```
      LHS {
        Op                    "SIN"
        LHS {
          Op                  "U"
          LHS {
            Op                "NUM"
            Value             1
          }
        }
      }
      RHS {
        Op                    "NUM"
        Value                 10
      }
    }
```

Block Type: `Gain` (example of a gain of `1:5`).

```
    Parameter {
      Name                    "Gain"
      Value                   [1, 2, 3, 4, 5]
      String                  "1:5"
      StringType              "Expression"
    }
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

Block Type: `Goto`

Virtual. Not written to *model*`.rtw` file.

Block Type: `Ground`

Virtual. Not written to *model*`.rtw` file.

Block Type: `HitCross` (example of a hit crossing block with an offset of 0).

```
Parameter {
  Name                    "HitCrossingOffset"
  Value                   [0]
  String                  "0"
  StringType              "Expression"
}
```

Block Type: `InitialCondition` (example of an initial condition block with an initial value of 1:5).

```
NumRWorkDefines           1
RWorkDefine {
  Name                    "FirstOutputTime"
  Width                   1
}
ParamSettings {
  Value                   [1, 2, 3, 4, 5]
}
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

Block Type: `Inport`

Virtual. Not written to *model*`.rtw` file.

Block Type: `Integrator` (shown below is a limited integrator configured with an internal initial condition of 0, an upper limit of `".75"`, and a lower limit of `"[–.25 0 –.75]"`).

```
Parameter {
  Name                    "InitialCondition"
  Value                   [0]
  String                  "0"
  StringType              "Expression"
}
Parameter {
  Name                    "UpperSaturationLimit"
  Value                   [0.75]
  String                  ".75"
  StringType              "Expression"
}
Parameter {
  Name                    "LowerSaturationLimit"
  Value                   [–0.25, 0, –0.75]
  String                  "[–.25 0 –.75]"
  StringType              "Expression"
}
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

`ParamSettings {`	
`ExternalReset`	`none`, `rising`, `falling`, or `either`
`InitialConditionSource`	`internal` or `external`
`LimitOutput`	`on` or `off`
`ShowSaturationPort`	`on` or `off`
`ShowStatePort`	`on` or `off`
`ExternalX0`	Only written when initial condition (`IC`) source is external. This is the initial value of the signal entering the IC port.
`}`	

Block Type: `Logic`

`ParamSettings {`	
`Operator`	`AND`, `OR`, `NAND`, `NOR`, `XOR`, or `NOT`
`}`	

Block Type: `Lookup` (example of a look up with `[-5:0]` for input values and `[0:5]` for output values).

`Parameter {`	The input values, `x`, to the function `y = f(x)`.
`Name`	`"InputValues"`
`Value`	`[-5, -4, -3, -2, -1, 0]`
`String`	`"[-5:0]"`
`StringType`	`"Expression"`
`}`	

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

`Parameter {`	The output values, `y`, of the function `y = f(x)`.
`Name`	`"OutputValues"`
`Value`	`[0, 1, 2, 3, 4, 5]`
`String`	`" [0:5]"`
`StringType`	`"Expression"`
`}`	
`Parameter {`	This is `(y(i+1)-y(i))/(x(i+1)-x(i))`
`Name`	`"Slopes"`
`Value`	`[1, 1, 1, 1, 1, 0]`
`String`	`""`
`StringType`	`"Computed"`
`}`	
`Parameter {`	This is the output of the block when the input to the block is zero.
`Name`	`"OutputAtZero"`
`Value`	`[5]`
`String`	`""`
`StringType`	`"Computed"`
`}`	

Block Type: `Lookup2d` (example of a look up with `1:2` and `1:3` for row and column input values and `[[4, 5 6]; [16, 18, 20]]` for output table values).

`Parameter {`	The "row" input values, `x`, to the function `z = f(x,y)`.

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

Name	"RowIndex"
Value	[1, 2]
String	"1:2"
StringType	"Expression"
}	
Parameter {	The "column" input values, y, to the function z = f(x,y).
Name	"ColumnIndex"
Value	[1, 2, 3]
String	"1:3"
StringType	"Expression"
}	
Parameter {	The "table" output values, z, to the function z = f(x,y).
Name	"OutputValues"
Value	Matrix(2,3) [[4, 5, 6]; [16, 18, 20];]
String	"[[4, 5, 6]; [16, 18, 20]]"
StringType	"Expression"
}	
Parameter {	This is the output of the block when the row input, x, to the block is zero.
Name	"OutputAtRowZero"
Value	[-8, -8, -8]
String	""

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

```
      StringType              "Computed"
    }
    ParamSettings {
      ColZeroTechnique        NormalInterp, AverageValue, or MiddleValue
      ColZeroIndex            0
    }
```

Block Type: `Math`

```
    ParamSettings {
      Operator                exp, log, 10^u, log10, square, sqrt, pow, reciprocal, hypot,
                              rem, or mod
    }
```

Block Type: `MATLABFcn`

There is no support for the MATLAB Fcn block in RTW.

Block Type: `Memory` (example of a memory block with an initial condition of 0).

```
    Parameter {
      Name                    "X0"
      Value                   [0]
      String                  "0"
      StringType              "Expression"
    }
    NumRWorkDefines           1
      Name                    "PrevU"
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

Width	2
}	
Block Type: MinMax	
ParamSettings {	
Function	min or max.
}	
Block Type: MultiPortSwitch	
No block specific records.	
Block Type: Mux	
Virtual. Not written to *model*.rtw file.	
Block Type: Outport	
ParamSettings {	
PortNumber	Port number as entered in the dialog box.
OutputLocation	Specified as Yi if root-level outport; otherwise specified as Bi.
OutputWhenDisabled	Only written when in an enabled subsystem and will be held or reset.
}	
Block Type: Probe	
ParamSettings {	
ProbeWidth	on or off

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

```
    ProbeSampleTime          on or off
  }
```

Block Type: `Product`

No block specific records.

Block Type: `Quantizer` (example of a quantizer block with a quantization interval of .5).

```
  Parameter {
    Name                     "QuantizationInterval"
    Value                    [.5]
    String                   "0.5"
  }
```

Block Type: `RandomNumber` (example of a random number block with a mean of 0, a variance of 1, and an initial seed of 0).

```
  Parameter {
    Name                     "Mean"
    Value                    [0]
    String                   "0"
    StringType               "Expression"
  }
  Parameter {
    Name                     "StandardDeviation"
    Value                    [1]
    String                   ""
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

```
      StringType              "Computed"
    }
    NumIWorkDefines           1
    IWorkDefine {
      Name                    "RandSeed"
      Width                   1
    }
    NumRWorkDefines           1
    RWorkDefine {
      Name                    "NextOutput"
      Width                   1
    }
```

Block Type: `RateLimiter` (example of a rate limiter block with a rising slew limit of 1, and a falling slew limit of –1).

```
    Parameter {
      Name                    "RisingSlewLimit"
      Value                   [1]
      String                  "1"
      StringType              "Expression"
    }
    Parameter {
      Name                    "FallingSlewLimit"
      Value                   [-1]
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

String	"-1"
StringType	"Expression"
}	
ParamSettings {	
Signal	Limit the rate of change of the output or input signal.
}	
NumRWorkDefines	1
RWorkDefine {	Used to keep track of last time, output and input.
Name	"PrevTYU"
Width	2*blockWidth+1 or 2*(2*blockWidth+1) where block width is the width of the input port after scalar expansion.
}	

Block Type: Reference

Will never appear in *model*.rtw.

Block Type: RelationalOperator

ParamSettings {	
Operator	One of ==, ~=, <, <=, >=, >.
}	

Block Type: Relay (example of a relay block with Switch on and off point of eps. Output is 1 when on and 0 when off).

Parameter {	
Name	"OnSwitchValue"

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

```
    Value                   [2.220446049250313e-16]
    String                  "eps"
    StringType              "Variable"
  }
  Parameter {
    Name                    "OffSwitchValue"
    Value                   [2.220446049250313e-16]
    String                  "eps"
    StringType              "Variable"
  }
  Parameter {
    Name                    "OnOutputValue"
    Value                   [1]
    String                  "1"
    StringType              "Expression"
  }
  Parameter {
    Name                    "OffSwitchValue"
    Value                   [0]
    String                  "0"
    StringType              "Expression"
  }
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

Block Type: `ResetIntegrator`

No support for the obsoleted Reset Integrator block.

Block Type: `Rounding`

```
ParamSettings {
  Operator                floor, ceil, round, or fix.
}
```

Block Type: `Saturate`

The following is an example of a saturation block configured with an upper limit of 0.5 and a lower limit of –.5,

```
Parameter {
  Name                    "UpperLimit"
  Value                   [0.5]
  String                  "–0.5"
}
Parameter {
  Name                    "UpperLimit"
  Value                   [0.5]
  String                  "–0.5"
}
```

Block Type: `Scope`

```
ParamSettings {
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

SaveToWorkspace	If scope is configured to save its data, then yes, otherwise no.
SaveName	Name of variable used to save scope data.
MaxRows	Maximum number of rows to save or 0 for no limit.
Decimation	Data logging interval.
}	

Block Type: Selector

Virtual. Not written to *model*.rtw file.

Block Type: S-Function.

Parameter {	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.
Name	Name is of the form P#Size.
Value	Value is dependent upon user data.
String	""
StringType	"Computed"
}	
Parameter {	
Name	Name is of the form P#.
Value	Value is dependent upon user data.
String	""
StringType	"Computed"
}	

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

`ParamSettings {`	
`FunctionName`	Name of S-function.
`FunctionType`	Type of S-function: `"M-File"`, `"C-MEX"`, or `"FORTRAN-MEX"`.
`DirectFeedthrough`	`yes` or `no`
`UsingUPtrs`	Is the C MEX S-function using `ssGetUPtrs(S)` or `ssGetU(S)`?
`InputContiguous`	`yes` or `no`
`SampleTimesToSet`	`M`-by-2 matrix of sample time indices indicating any sample times specified by the S-function in `mdlInitializeSampleTimes`, which get updated. The first column is the S-function sample time index, and the second 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.
`DynamicallySizedVectors`	Vector containing any of: `"U"`, `"Y"`, `"Xc"`, `"Xd"`, `"RWork"`, `"IWork"`, or `"PWork"`. For example `["U", "Y"]`.
`}`	
`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-3: Model.rtw File Contents — Block Specific Records (Continued)

Block Type: `SignalGenerator` (example of a signal generator with an amplitude of 1 and a frequency of 1).

```
Parameter {
  Name                  "Amplitude"
  Value                 [1]
  String                "1"
  StringType            "Expression"
}
Parameter {
  Name                  "Frequency"
  Value                 [1]
  String                "1"
  StringType            "Expression"
}
ParamSettings {
  WaveForm              sine, square, or sawtooth
  TwoPi                 6.283185307179586
  StringType            "Expression"
}
```

Block Type: `Signum`

No block specific records.

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

Block Type: `Sin` (The following is an example for the Sine Wave block configured with a discrete sample time of 1 second):

```
Parameter {
  Name                    "Amplitude"
  Value                   [1]
  String                  "1"
  StringType              "Expression"
}
Parameter {
  Name                    "Frequency"
  Value                   [1]
  String                  "1"
  StringType              "Expression"
}
Parameter {
  Name                    "Phase"
  Value                   [0]
  String                  "0"
  StringType              "Expression"
}
Parameter {
```

This is a discrete sine coefficient and is only written when the Sine Wave block has a discrete sample time.

```
  Name                    "sin_h"
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

Value	[0.009999833334166664]
String	""
StringType	"Computed"
}	
Parameter {	This is a discrete sine coefficient and is only written when the Sine Wave block has a discrete sample time.
Name	"cos_h"
Value	[0.9999500004166653]
String	""
StringType	"Computed"
}	
Parameter {	This is a discrete sine coefficient and is only written when the Sine Wave block has a discrete sample time.
Name	"sin_phi"
Value	[–0.009999833334166664]
String	""
StringType	"Computed"
}	
Parameter {	This is a discrete sine coefficient and is only written when the Sine Wave block has a discrete sample time.
Name	"cos_phi"
Value	[0.9999500004166653]
String	""
StringType	"Computed"

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

```
}
```

Block Type: `StateSpace`

See Model.rtw File Contents — Linear Block Specific Records on page A-51.

Block Type: `Sum`

```
ParamSettings {
  Inputs                  A vector of the form ["+", "+", "-"] corresponding to the
                          configuration of the block.
}
```

Block Type: `SubSystem`

```
ParamSettings {
  SystemIdx               Index of this system in the model.rtw file.
  StatesWhenEnabling      held or reset. Only written if enable port is present.
  TriggerBlock            Block index of TriggerPort block in system.
  SystemContStates        Specified as [N,I] where N is the number of continuous states
                          and I is the index into the state vector, X.
}
```

Block Type: `Switch` (Example of a switch with a threshold of 0).

```
Parameter {
  Name                    "Threshold"
  Value                   [0]
  String                  "0"
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

```
    StringType              "Expression"

  }
```

Block Type: `ToFile`
The following is an example of a ToFile block configured with a filename of `untitled.mat`, and a matrix name of `ans`. The `IWork` contains two fields; one is for tracking the number of rows written (`Count`) and the other is for determining when to log the data at the input (`Decimation`).

```
  NumIWorkDefines           2
  IWorkDefine {
    Name                    "Count"
    Width                   1
  }
  IWorkDefine {
    Name                    "Decimation"
    Width                   1
  }
  NumRWorkDefines           1
  RWorkDefine {
    Name                    "FilePtr"
  }
  ParamSettings {
    Filename                "untitled.mat"
    MatrixName              "ans"
    Decimation              1
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

```
  }
```

Block Type: `ToWorkspace`

```
  ParamSettings {
    VariableName          Name of variable used to save scope data.
    Buffer                Maximum number of rows to save or 0 for no limit.
    Decimation            Data logging interval.
    InputContiguous       yes or no
  }
```

Block Type: `Terminator`

Virtual. Not written to *model*`.rtw` file.

Block Type: `TransferFcn`

See Model.rtw File Contents — Linear Block Specific Records on page A-51.

Block Type: `TransportDelay` (example of a transport delay with a time delay of 1, an initial output of 0, and an initial buffer size of 1024).

```
  Parameter {
    Name                  "DelayTime"
    Value                 [1]
    String                "1"
    StringType            "Expression"
  }
  ParamSettings {
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

InitialInput	[0]
BufferSize	[1024]
}	
NumIWorkDefines	1
IWorkDefine {	
Name	"BufferIndices"
Width	4
}	
NumPWorkDefines	1
PWorkDefine {	
Name	"TUbuffer"
Width	2
}	

Block Type: `TriggerPort`

ParamSettings {	Only written if the number of output ports is one.
TriggerType	This will be one of "rising", "falling", "either", or "function-call".
}	

Block Type: `Trigonometry`

ParamSettings {	
Operator	sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, or tanh
}	

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

Block Type: `UniformRandomNumber` (example of a uniform random number block with minimum of −1, maximum of 1, initial seed of 0).

```
Parameter {
  Name                  "Minimum"
  Value                 [-1]
  String                "-1"
  StringType            "Expression"
}
Parameter {
  Name                  "MaxMinusMin"
  Value                 [2]
  String                ""
  StringType            "Computed"
}
NumIWorkDefines         1
IWorkDefine {
  Name                  "RandSeed"
  Width                 1
}
NumRWorkDefines         1
RWorkDefine {
  Name                  "NextOutput"
  Width                 1
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

```
  }
```

Block Type: `UnitDelay` (example of a unit delay with an initial condition of 0).

```
  Parameter {
    Name                    "X0"
    Value                   [0]
    String                  "0"
    StringType              "Expression"
  }
```

Block Type: `VariableTransportDelay` (example of a variable transport delay with a maximum delay of 10, an initial input of 0, and a buffer size of 1024).

```
  Parameter {
    Name                    "Maximum"
    Value                   [10]
    String                  "10"
    StringType              "Expression"
  }
  ParamSettings {
    InitialInput            [0]
    BufferSize              [1024]
  }
  NumIWorkDefines           1
  IWorkDefine {
```

Table A-3: Model.rtw File Contents — Block Specific Records (Continued)

```
      Name                    "BufferIndices"
      Width                   4
    }
    NumPWorkDefines           1
    PWorkDefine {
      Name                    "TUbuffer"
      Width                   2
    }
```

Block Type: `Width`

No block specific records.

Block Type: `ZeroPole`

See Model.rtw File Contents — Linear Block Specific Records on page A-51.

Block Type: `ZeroOrderHold`

No block specific records.

The following table describes the block specific records written for the Simulink linear blocks.

Table A-4: Model.rtw File Contents — 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.

Table A-4: Model.rtw File Contents — Linear Block Specific Records (Continued)

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 {	
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.

Table A-4: Model.rtw File Contents — Linear Block Specific Records (Continued)

`ColIdxOfNonZeroD`	Column index of the nonzero elements in the `D` matrix.
`}`	

B

Target Language Compiler Error Messages

This appendix lists and describes error messages generated by the Target Language Compiler. Use this reference to:

- Confirm that an error has been reported.
- Determine possible causes for an error.
- Determine possible ways to correct an error.

%closefile or %selectfile argument must be a valid open file

When using `%closefile` or `%selectfile`, the argument must be a valid file variable opened with `%openfile`.

%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:** This directive causes the Target Language Compiler to terminate regardless of the `-m`*number* command line option.

%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:** `%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:** `%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" in Chapter 2 for more information.

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.

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]
```

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
FILE_EXISTS	GENERATE_TYPE
FORMAT	IDNUM
GENERATE	SYSNAME

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.

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

The name of a function cannot be defined prior to encountering the definition of the function.

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 );
```

Cannot convert string *string* to a number

Cannot convert the string to a number.

Cannot redefine existing symbol *identifier* (use %undef)

You cannot redefine a macro without using `%undef` first.

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.

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 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

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 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.

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`.

Incorrect number of arguments to a macro (*number* expected)

When invoking a function-like macro, too many arguments were passed to it. The extra arguments were ignored.

Indices must be constant integral numbers

An index used in a `[]` expression must be an integral number.

Invalid type for unary *operator*

This error occurs for the following operators under the given conditions:

Operator	Reason for error
!	Operand to the logical not operator (!) must be a Number, Real, or Boolean.
-	Operand to the unary negation operator (-) must be a Number or Real.
+	Operand to the unary plus operator (+) must be a Number or Real.
~	Operand to the bitwise negation operator (~) must be a Number.

It is illegal to return functions or macros from a function

A function or macro 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
```

Only macros, function calls, and built-in functions can be used with the function call syntax ( )

The function call syntax ( ) can only be used with functions. This error means that you attempted to call a nonfunction. For example:

```
%assign x = 1
%assign y = x(1)
```

This code produces this error because you cannot use the function call syntax for nonfunction variables.

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 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.

Syntax error

The indicated line contains a syntax error, See Chapter 2, "Working with the Target Language," for information on the syntax.

Syntax error detected in EXISTS function called with "*string*"

The `EXISTS` function parses and evaluates the string passed to it. The function reports this error when it is unable to parse the input string successfully. To better diagnose the error, you can try to define the symbol and then type the identical expression inside an expansion directive. For example:

```
%if EXISTS( "x[100].y" )
%% If this fails, try
%<x[100].y>
%% In order to receive a better diagnosis of the problem.
```

The %break directive can only appear within a %foreach, %for, %roll, or %switch statement

The `%break` directive can only be used in a `%foreach`, `%for`, `%roll`, or `%switch` statement.

The %case and %default directives can only be used within the %switch statement

A `%case` or `%default` directive can only appear within a `%switch` statement.

The %codeblock, %endcodeblock, and %generate directives are obsolete; use %function and %<GENERATE( ) instead.

All of the directives listed here are obsolete; use `%<GENERATE( )>` instead.

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 :: operator can only be used within a function (Warning)

The :: operator (used to specify global scope within a function) should not be used outside of a function body.

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 %include must be a valid string

The argument to the input file control directive must be a valid string.

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 (page 2-18)
%foreach	%endforeach	Looping (page 2-19)
%roll	%endroll	Loop Rolling (page 2-22)
%with	%endwith	Scoping directive (page 2-41)
%switch	%endswitch	Switch directive (page 2-18)
%function	%endfunction	Function declaration directive (page 2-44)

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 *directive* block that begins on this line has no corresponding end

This error message indicates that a block-scoped directive (`%if`, `%with`, `%foreach`, `%for`, `%roll`, `%function`, or `%switch`) had no corresponding end directive (`%endif`, `%endwith`, `%endforeach`, `%endfor`, `%endroll`, `%endfunction` or `%endswitch`). **Note:** You must end blocks in the Target Language Compiler in the same order that you begin them. The most common cause of this error is improperly nested constructs, for example:

```
%if x == 3
    %with scope      %% Error on this line
%endif
%endwith
```

The FEVAL() function can accept only 2-dimensional arrays from MATLAB, not *identifier* 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.

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 Chapter 2 for a complete discussion of the `%roll` construct.

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 "GENERATE and GENERATE_TYPE Functions" on page 2-25 for more information and examples.

The GENERATE function requires at least two 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 three 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(), and ISFINITE() functions expect a real valued argument

These functions expect a `Real` 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 rows seen in the matrix (2). Either change the number of rows 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 (i.e., 1) did not match the number of rows seen in the matrix (i.e., 2). Either change the number of rows in the matrix, or change the matrix declaration.

The *operator* operator only works on numeric arguments

The arguments to the following operators both must be either Number or Real: `<`, `<=`, `>`, `>=`, `+`, `-`, `*`, `/`. In addition, the `FORMAT` built-in function expects either a Number or Real argument.

The *operator* operator only works on integral arguments

The `&`, `^`, `|`, `<<`, `>>` and `%` operators only work on numbers.

The *operator* operator only works on Boolean arguments

The `&&` and `||` operators work on Boolean values only.

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 Chapter 2 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 Chapter 2 for a complete discussion of the `%roll` construct.

The specified index (*index*) was out of the range 0 to number-of-elements – 1

This error occurs when indexing into any nonscalar beyond the end of the variable. For example:

```
%assign x = [1 2 3]
%assign y = x[3]
```

This example would cause this error. Remember, in the Target Language Compiler, array indices start at 0 and go to the number of elements minus 1.

The STRINGOF built-in function expects a vector of numbers as its argument

The `STRINGOF` function expects a vector of numbers. The function treats each number as the ASCII value of a valid character.

The SYSNAME built-in function expects an input string of the form <xxx>/yyy

The `SYSNAME` function takes a single string of the form `<xxx>/yyy` as it appears in the `.rtw` file and returns a vector of two strings `xxx` and `yyy`. If the input argument does not match this format, it returns this error.

The threshold on a %roll statement must be a single number

When using `%roll`, the roll threshold specified must be a single number. See Chapter 2 for a complete discussion of the `%roll` construct.

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 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 page 2-25 for more information and examples on using the `GENERATE` built-in function.

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 may:

- 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*

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 on pages 2-10 through 2-13.

Unrecognized directive "*directive-name*" seen

An illegal % directive was encountered. The valid directives are:

Table B-1: Valid Directives

%assign	%for
%break	%foreach
%case	%function
%closefile	%generatefile
%continue	%if
%default	%implements
%define	%include
%else	%language
%elseif	%openfile
%endbody	%realformat
%endfor	%return
%endforeach	%roll
%endfunction	%selectfile

Table B-1: Valid Directives (Continued)

%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 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[N]	Specify the verbose level to be `N` (1 by default).
–I<path>	Specify a search path to look for `%include` and `%generate` files.
–m[N|a]	Specify the maximum number of errors (`a` is all) default is 5.
–O<path>	Specify the path used to create output files.
–d[g|n|o]	Specify debug mode (`generate`, `normal`, or `off`).
–a<ident>=<expression>	Assign a variable to a specified value.

A command line problem has occurred. The error message contains a list of all of the available options.

Value of *type* type cannot be compared

The specified type (i.e., scope) cannot be compared.

Values of *type* type cannot be expanded

The specified type cannot be used on an expansion directive. Files and scopes cannot be expanded.

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
```

C

Target Language Compiler Library Error Messages

This appendix lists and describes error messages that may be generated when working with the Target Language Compiler libraries. In this appendix, the error messages are grouped by the libraries because they are reported by library. For example, this error message is generated by the `rolllib.tlc` library.

```
rolllib.tlc:345: There are no modes to roll in ?? block: test1
```

Each error message contains three components:

- Error message text
- Function(s) that generates the error message (in parentheses)
- Description of the error message

blkiolib.tlc Error Messages

Invalid map source (*mappingSource*) specified for *Type* block: *Name*

(`LibMapSignalSource`) Real-Time Workshop does not generate a mapping matrix for the specified source. Valid map sources are:

Map Source	Description
`U`	External inputs map
`X`	States map
`B`	Block I/O map
`G`	Ground

Invalid port number (*portNum*) specified for *Type* block: *Name*

(`LibDataOutputPortWidth`, `LibDataInputPortWidth`) The specified output port does not exist for this block.

blocklib.tlc Error Messages

Don't know how to roll IWork for *Type* block: *Name*

(`LibBlockIWork`) In order to roll a block's `IWork` it must first be defined with `LibDefineIWork`.

Don't know how to roll PWork for *Type* block: *Name*

(`LibBlockPWork`) In order to roll a block's `PWork` it must first be defined with `LibDefinePWork`.

Don't know how to roll RWork for *Type* block: *Name*

(`LibBlockRWork`) In order to roll a block's `RWork` it must first be defined with `LibDefineRWork`.

Invalid control port (*id*) specified for *Type* block: *Name*

(`LibControlPortIndexNumber`) Valid IDs are `enable`, `trigger`, and `function-call`.

hookslib.tlc Error Messages

Add root initialization code with LibMdlStartCustomCode

(`LibSystemInitializeCustomCode`) The root system initialization function is `MdlStart`. Therefore, use `LibMdlStartCustomCode` for placement of root system initialization code.

Invalid location: *location*

(`LibHeaderFileCustomCode`, `LibPrmFileCustomCode`, `LibRegFileCustomCode`, `LibMdlStartFcnCustomCode`, `LibMdlTerminateCustomCode`, `LibRegFcnCustomCode`, `LibSystemInitializeCustomCode`, `LibSystemOutputCustomCode`, `LibSystemUpdateCustomCode`, `LibSystemDerivativeCustomCode`, `LibSystemEnableCustomCode`, `LibSystemEnableCustomCode`)
Valid locations for code placement:
header
trailer

Invalid location: *location*

(`LibSourceFileCustomCode`)
Valid locations for code placement:
header

System *system.Name* does not have Derivatives function

(`LibSourceFileCustomCode`) The subsystem's derivative function is eliminated if there are no residing states.

System *system.Name* does not have Disable function

(`LibSystemDisableCustomCode`) The subsystems types that have a disable function are:
Purely enable subsystems
Enable with trigger subsystems

The system types that do not have a disable function are:
Function-call
Trigger
Root

System *system.Name* does not have Enable function

(`LibSystemEnableCustomCode`) The subsystems types that have an enable function are:
Purely enable subsystems
Enable with trigger subsystems

The system types that do not have an enable function are:
Function-call
Trigger
Root

paramlib.tlc Error Messages

Loop rolling not supported for *param.Name* in *Type* block *Name*

(`LibBlockMatrixParamter`, `LibBlockMatrixParameterAddr`) Loop rolling of matrix parameters is not supported. The arguments are passed into this routine as protection for future support of this feature.

Parameter *param.Name* must be of type Matrix

(`LibBlockMatrixParamter`, `LibBlockMatrixParameterAddr`) This routine only works for matrix parameters. Use `LibBlockParameter` to access a vector or scalar block parameter.

Type block *Name* must access *param.Name* via LibBlockMatrixParameter

(`LibBlockParameter`) This routine does not work for matrix parameters. Use `LibBlockMatrixParamter` when accessing a block's matrix parameter.

Type block *Name* must access *param.Name* via LibBlockMatrixParameterAddr

(`LibBlockParameterAddr`) This routine does not work for matrix parameters. Use `LibBlockMatrixParamterAddr` when accessing a block's matrix parameter.

rolllib.tlc Error Messages

1-- The inputs for *Type* block *Name* are not rollable, or do not exist

(`LibDeclareRollVariables`) The variable `rollVars` contains `U` (declare all inputs), but the block does not have any inputs to declare.

2-- u*uIdx* for *Type* block *Name* is not rollable, or does not exist

(`LibDeclareRollVariables`) The variable `rollVars` contains `ui` (declare input `i`), but input `i` does not exist.

3-- The outputs for *Type* block *Name* are not rollable

(`LibDeclareRollVariables`) The variable `rollVars` contains `Y` (declare all outputs), but the block does not have any outputs to declare. The Outport block is the only block that is allowed to do this.

4-- y*yIdx* for *Type* block *Name* is not rollable

(`LibDeclareRollVariables`) The variable `rollVars` contains `yi` (declare output `i`), but the block does not have any outputs to declare. The Outport block is the only block that is allowed to do this.

5-- There are no discrete states to roll in *Type* block: *Name*

(`LibDeclareRollVariables`) The variable `rollVars` contains `xd` or `Xd` (declare discrete states), but the block does not have any discrete states to declare.

6-- There are no continuous states to roll in *Type* block: *Name*

(`LibDeclareRollVariables`) The variable rollVars contains `xc` or `Xc` (declare continuous states), but the block does not have any continuous states to declare.

7-- There are no parameters to roll in *Type* block: *Name*

(`LibDeclareRollVariables`) The variable `rollVars` contains `P` (declare all parameters), but the block does not have any parameters to declare.

8-- Unable to declare roll variable p_*name* for *Type* block *Name*

(`LibDeclareRollVariables`) The variable `rollVars` contains `<param>/p` (declare parameter `p`), but `p` is not a valid block parameter.

9-- Unable to roll RWork for *Type* block *Name*. RWork must be defined

(`LibDeclareRollVariables`) The variable `rollVars` contains `RWork` (declare all real-work), but the block does not have any real-work to declare.

10-- Unable to declare roll variable rw_*name* for *Type* block *Name*

(`LibDeclareRollVariables`) The variable `rollVars` contains `<rwork>/r` (declare real-work `r`), but `r` is not a valid real-work name.

11-- Unable to roll IWork for *Type* block *Name*. IWork must be defined

(`LibDeclareRollVariables`) The variable `rollVars` contains `IWork` (declare all integer-work), but the block does not have any integer-work to declare.

12-- Unable to declare roll variable iw_*name* for *Type* block *Name*

(`LibDeclareRollVariables`) The variable `rollVars` contains `<iwork>/i` (declare integer-work `i`), but `i` is not a valid integer-work name.

13-- Unable to roll PWork for *Type* block *Name*. PWork must be defined

(`LibDeclareRollVariables`) The variable `rollVars` contains `PWork` (declare all pointer-work), but the block does not have any pointer-work to declare.

14-- Unable to declare roll variable pw_*name* for *Type* block *Name*

(`LibDeclareRollVariables`) The variable `rollVars` contains `<pwork>/p` (declare pointer-work `p`), but `p` is not a valid pointer-work name.

15-- There are no modes to roll in *Type* block: *Name*

(`LibDeclareRollVariables`) The variable `rollVars` contains `Mode` (declare block modes), but the block does not have a modes vector to declare.

16-- There are no previous zero-crossings to roll in *Type* block: *Name*

(`LibDeclareRollVariables`) The variable `rollVars` contains `PZC` (declare block previous zero-crossing vector), but the block does not have a zero-crossing vector to declare.

17-- There are no data store memory values to roll in *Type* block: *Name*

(`LibDeclareRollVariables`) The variable `rollVars` contains `DSM` (declare data store memory variables), but the block does not have any data store memory variables to declare.

18-- Unknown roll variable (*rollVarArg*) specified by *Type* block: *Name*

(`LibDeclareRollVariables`) The `rollVars` argument is not recognized.

utillib.tlc Error Messages

Invalid matrix size (*nRows* x *nCols*) for *Type* block: *Name*

(`LibOptionalMatrixWidth`) Either `nRows` or `nCols` is less than 1.

Invalid vector length (*length*) specified for *Type* block: *Name*

(`LibOptionalVectorWidth`) The value of `length` is less than 1.

Invalid zc direction (*direction*) specified for *Type* block: *Name*

(`LibConvertZCDirection`) RTW zero-crossings are either `Rising`, `Falling`, or `Any`. An unrecognized zero-crossing was specified.

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

Z