SIMULINK®

Dynamic System Simulation for MATLAB®

Modeling

Simulation

Implementation

Target Language Compiler Reference Guide

Version 1.2

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
June 1998 Second printing Target Language Compiler 1.1
January 1999 Online revision for Target Language Compiler 1.2 (R 11)

Contents

1 Using the Target Language Compiler

2 Working with the Target Language

3 Target Language Files

4 Writing Target Language Files: A Tutorial

5 Target Language Compiler Function Library Reference

A model.rtw

1

Using the Target Language Compiler

Using the Target Language Compiler

Introduction

The Target Language Compiler™ is a tool that is included with Real-Time Workshop® and enables you to customize the C or Ada 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.

Note This book describes the Target Language Compiler, its files, and how to use them together. This information is provided for those users who need to customize target files in order to generate specialized output. Or, in some cases, for users who want to inline S-functions so as to improve the performance of the generated code.

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 or Ada code.

In order to create a target-specific application, Real-Time Workshop also requires a template makefile that specifies the appropriate C or Ada 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 or Ada 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 Chapter 4, "Writing Target Language Files: A Tutorial." 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), header files (*model*.h and *model*_export.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.

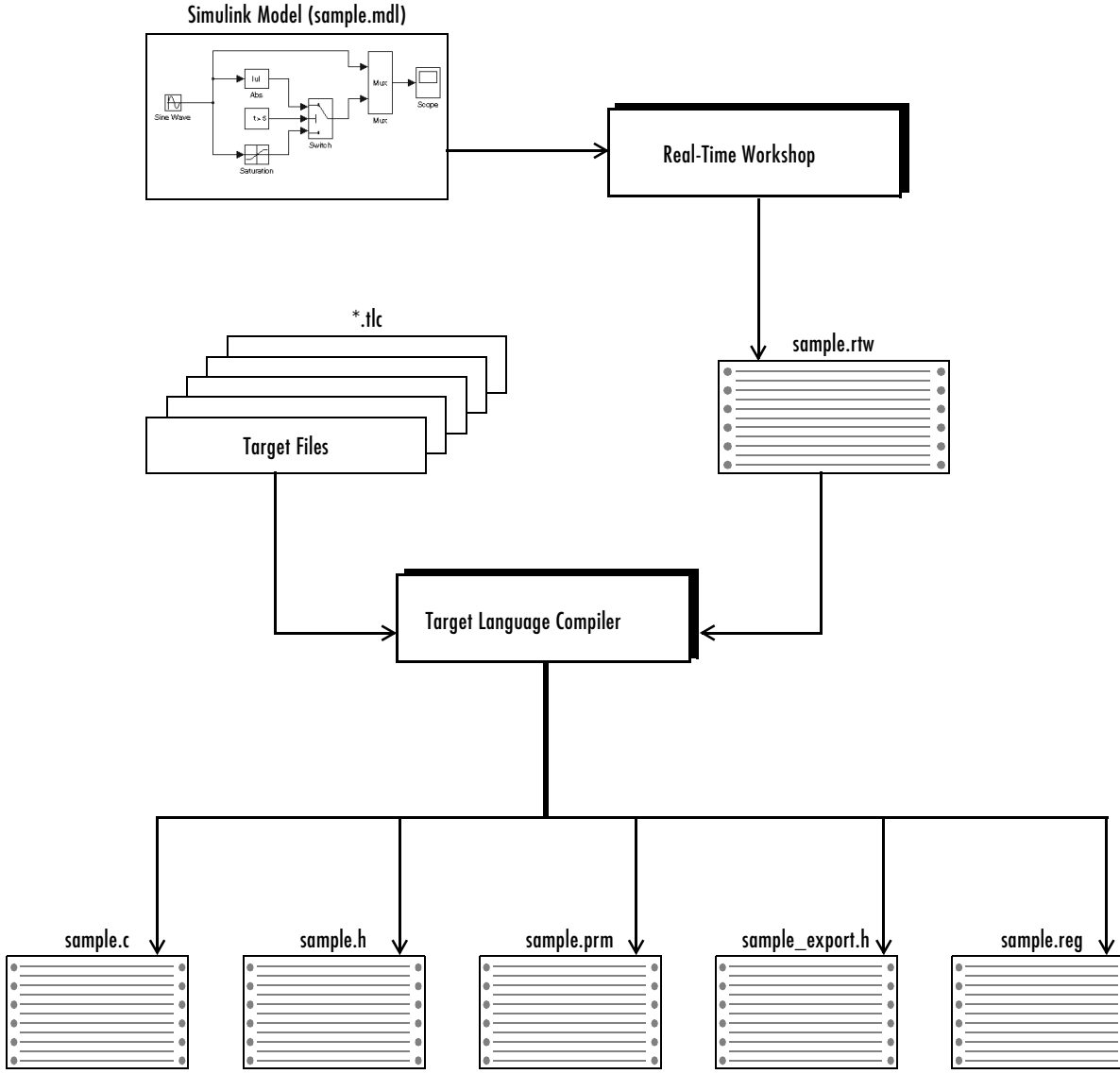
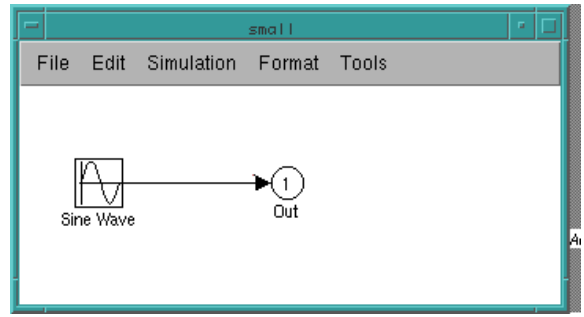


Figure 1-1: The Target Language Compiler Process

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 Real-Time Workshop'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. Create this model and save it as `small.mdl`.



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  "3.0 $Date: 1998/12/18 22:48:41 $"
  ModelVersion  "1.1"
  GeneratedOn  "Thu Dec 31 10:22:43 1998"
  Solver  FixedStepDiscrete
  SolverType  FixedStep
  StartTime  0.0
  StopTime  10.0
                      <Parameters omitted>

  BlockOutputs {
      BlockOutputDefaults {
        TestPoint      no
        StorageClass      Auto
        StorageTypeQualifier    ""
        IdentifierScope      "top-level"
        Invariant      no
        InitialValue      []
        DataTypeIdx      0
        ComplexSignal      no
        SigSrc      []
        SigLabel      ""
        SigConnected      all
      }
      ReusedBlockOutputDefaults {
        SigLabel      ""
        SigConnected      all
      }
      MergedBlockOutputDefaults {
        SigLabel      ""
        SigConnected      all
      }
      NumBlockOutputs    1
      BlockOutput {
        Identifier      Sine_Wave
        SigIdx      [0, 1]
        IdentifierScope      "fcn-level"
```

```
    SigSrc      [0, 0, 0]
  }
  DataTypeTransitions {
    NumTransitions      0
  }
}
BlockOutputsMap  [0]
DWorkRecords {
  DWorkRecordDefaults {
    DataTypeIdx      0
    ComplexSignal      no
    UsedAsDState      no
  }
  NumDWorkRecords    0
}
BlockDefaults {
               <Block defaults omitted>

System {
  Type    root
  Name    "<Root>"
  Identifier    root
  NumBlocks    2
  BlocksIdx    0
  NumVirtualOutportBlocks 0
  VirtualOutportBlocksIdx 2
  NumTotalBlocks    2
  Block {
    Type      Sin
    Name      "<Root>/Sine Wave"
    Identifier      Sine_Wave
    TID      0
    RollRegions      [0]
    NumDataOutputPorts      1
    DataOutputPortIndices   [0]
    Parameters      [3, 3]
                <Parameters omitted>

Block {
  Type      Outport
```

```
      Name      "<Root>/Out1"
      Identifier      Out1
      TID      0
      RollRegions      [0]
      NumDataInputPorts      1
      DataInputPort {
    Width       1
    SignalSrc[B0]
      }
      ParamSettings {
    PortNumber1
    OutputLocationY0
    SpecifyICno
      }
     }
     EmptySubsysInfo {
      NumRTWdatas      0
     }
   }
   ModelParameters {
     NumParameters    3
     NumInrtP    3
     NumInlinedUnlessRolled  0
     NumExportedGlobal    0
     NumImportedExtern    0
     NumImportedExternPointer 0
     ParameterDefaults {
       Tunable      no
       StorageClass      Auto
       TypeQualifier      ""
     }
     Parameter {
       Identifier      Sine_Wave_Amp
       Tunable      yes
       ReferencedBy      Matrix(1,3)
[[0, 0, 0];]
     }
     Parameter {
       Identifier      Sine_Wave_Freq
       Tunable      yes
```

```
      ReferencedBy      Matrix(1,3)
[[0, 0, 1];]
    }
    Parameter {
      Identifier      Sine_Wave_Phase
      Tunable      yes
      ReferencedBy      Matrix(1,3)
[[0, 0, 2];]
    }
  }
  BlockParamChecksum  Vector(4)
["2276165574U", "2458155296U", "1231694984U", "899320395U"]
  ModelChecksum  Vector(4)
["2832348324U", "1314814527U", "2455571061U", "2100822809U"]
```

To use the Target Language Compiler on `small.rtw` to generate all associated Real-Time Workshop code, enter:

```
tlc –r small.rtw matlabroot/rtw/c/grt/grt.tlc
    –Imatlabroot/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 Real-Time

Workshop code. The generated output consists of the files: `small.c`, `small.h`, `small_export.h`, `small.prm`, and `small.reg`.

Table 1-1: Files that the Target Language Compiler Generates from small.mdl

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`.
`small_export.h`	Header file included in any user source code that interfaces with Real-Time Workshop generated code

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 Real-Time Workshop code (*model*`.rtw`) produced by Simulink into target-specific code.

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

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

Model-Wide Target Files and System Target Files

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

The system target file is the *entry point* for the TLC program, which is analogous to the `main()` routine of a C program. System target files oversee the entire code generation process. For example, the system target file, `grt.tlc`, sets up some variables for `codegenentry.tlc`, which is the entry point into the Real-Time Workshop target files. For a complete list of available system target files for the Real-Time Workshop, see Chapter 3 of the *Real-Time Workshop User's Guide*.

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

Table 1-2: Model-Wide Target Files for Static Real-Time, Malloc (dynamic) Real-Time, Embedded-C and RTW S-Function Applications

Model-Wide Target File	Code Format	Purpose
`ertautobuild.tlc`	Embedded-C	Includes *`model`*`_export.h` in the generated code

`srtbody.tlc`
`mrtbody.tlc`
`ertbody.tlc`
`sfcnbody.tlc`
	Static real-time
	Malloc real-time
	Embedded-C
	RTW S-function
		Creates the source file, *`model`*`.c`, which contains the procedures that implement the model

`srtexport.tlc`
`mrtexport.tlc`
`ertexport.tlc`
`sfcnexport.tlc`
	Static real-time
	Malloc real-time
	Embedded-C
	RTW S-function
		Creates the header file *`model`*`_export.h`, which defines access to external parameters and signals (all formats)

`srthdr.tlc`
`mrthdr.tlc`
`erthdr.tlc`
`sfcnhdr.tlc`
	Static real-time
	Malloc real-time
	Embedded-C
	RTW S-function
		Creates the header file *`model`*`.h`, which defines the data structures used by *`model`*`.c`. The data structures defines include BlockOutputs, Parameter, External Inputs and Outputs, and the various work structures. The instances of these structures are declared in *`model`*`.c` (all formats).

`srtlib.tlc`
`mrtlib.tlc`
`ertlib.tlc`
`sfclib.tlc`
	Static real-time
	Malloc real-time
	Embedded-C
	RTW S-function
		Contains utility functions used by the other model-wide target files (all formats)

Table 1-2: Model-Wide Target Files for Static Real-Time, Malloc (dynamic) Real-Time, Embedded-C and RTW S-Function Applications (Continued)

Model-Wide Target File	Code Format	Purpose

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

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

`srtparam.tlc`
`mrtparam.tlc`
`ertparam.tlc`
`sfcnparam.tlc`
	Static real-time
	Malloc real-time
	Embedded-C
	RTW S-function
		Creates the source file *`model`*`.prm`, which is included by the *`model`*`.c` file to declare instances of the various data structures defined in *`model`*`.h` (all formats)

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

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

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

Table 1-3: Model-Wide Target Files for Ada

Model-Wide Target File	Purpose
`adabody.tlc`	Generates the *`model`*`.adb` file, which contains the package body for the model.
`adalib.tlc`	Contains utility functions used by the other model-wide target files
`adaspec.tlc`	Generates the *`model`*`.ads` file, which contains the package specification for the model, and the package specifications required for auto-building simulation and real-time targets: • `register.ads` • `register2.ads` • `rt_engine-rto_data.ads`
`adatypes.tlc`	Generates the *`model`*`_types.ads` file, which contains the data structures required by the model. Note that all data declaration instances are declared in the *`model`*`.adb` file.
`adawide.tlc`	The entry point for Ada code format. It produces these files: • *`model`*`.ads` • *`model`*`.adb` • *`model`*`_types` • `register.ads` • `register2.ads` • `rtengine-rto_data.ads`

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:

- Chapter 1, “Using the Target Language Compiler,” provides overview information of the Target Language Compiler and its files.
- Chapter 2, “Working with the Target Language,” provides a complete description of the constructs used to create target language files and general coding guidelines.
- Chapter 3, “Target Language Files,” describes the structure of target files and includes a set of recommended coding conventions.
- Chapter 4, “Writing Target Language Files: A Tutorial,” describes the process of customizing target files and inlining S-functions, and concludes with a discussion of loop rolling.
- Chapter 5, “Target Language Compiler Function Library Reference,” Provides complete descriptions of all functions used to create block target files and inline S-functions.
- Appendix A, “model.rtw,” contains a complete description of the *model*`.rtw` file generated by Real-Time Workshop build procedure.
- Appendix B, “Target Language Compiler Error Messages,” contains the error messages generated by the Target Language Compiler and their descriptions.
- Appendix C, “Block Target File Mapping,” provides a complete list of Simulink block types and the associated target file that the Target Language Compiler uses to generate the corresponding code.

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. use the Target Language Compiler if you need to:

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

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 Target Language Compiler directives and constructs to modify existing target files or create new ones, depending on your needs. Chapter 2 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 Real-Time Workshop 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.

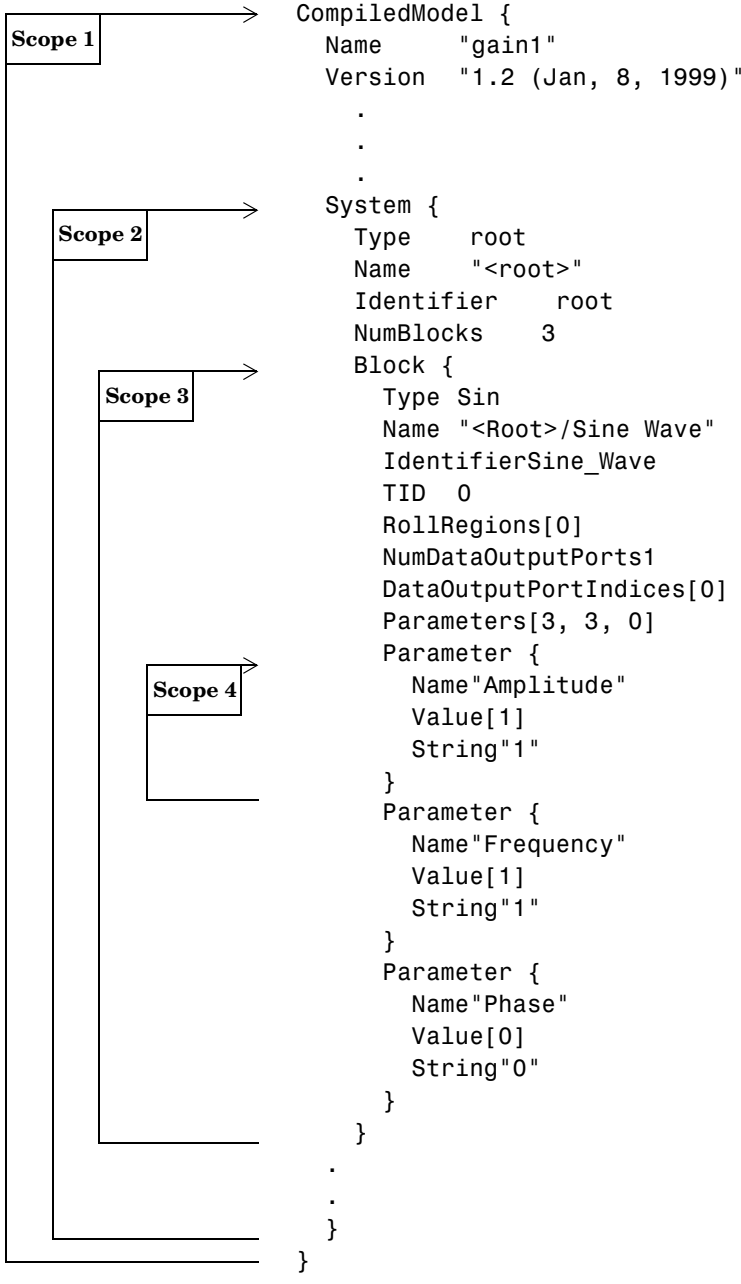
A record is specified as:

```
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`.


Scope 1
Scope 2
Scope 3
Scope 4
CompiledModel {
Name "gain1"
Version "1.2 (Jan, 8, 1999)"
.
.
.
System {
Type root
Name "<root>"
Identifier root
NumBlocks 3
Block {
Type Sin
Name "<Root>/Sine Wave"
IdentifierSine_Wave
TID 0
RollRegions[0]
NumDataOutputPorts1
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
```

Indexing begins at 0, so to access `Name` within the second `Parameter` record within `Block` within `System` within `CompiledModel`, you would use

```
CompiledModel.System.Block.Parameter[1].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 Target Language Compiler 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`
	`%flushfile identifier`
	
	`%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`

Callback to MATLAB	`%MATLAB identifier ( optional-arguments )`

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
```

Note Use `\` to suppress line feeds to the output and `...` to indicate line continuation.

Target Language Values

All expressions in Target Language Compiler must be of some type. 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.
`"Complex"`	`3.0+5.0i`	A 64-bit double-precision complex number.
`"Complex32"`	`3.0F+5.0Fi`	A 32-bit single-precision complex number.
`"File"`	`%openfile x`	String buffer opened with `%openfile`.
`"File"`	`%openfile x = "out.c"`	File opened with `%openfile`.
`"Function"`	`%function foo…`	A user-defined function.
`"Gaussian"`	`3+5i`	A 32-bit integer imaginary number.

Table 2-2: Target Language Values (Continued)

Value Type String	Example	Description
`"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.
`"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`	A 32-bit 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 64-bit double-precision floating-point number (including exponential notation).

Table 2-2: Target Language Values (Continued)

Value Type String	Example	Description
`Real32`	`3.14159F`	A 32-bit single-precision floating-point number
`"Scope"`	`Block { … }`	A block-scope.
`"Special"`	`N/A`	A special built-in function, such as `FILE_EXISTS`.
`"String"`	`"Hello, World"`	ASCII character strings. In all contexts, two strings in a row are concatenated to form the final value, as in `"Hello, " "World"`, which is combined to form `"Hello, World"`. These strings include all of the ANSI C standard escape sequences such as `\n`, `\r`, `\t`, etc.
`"Subsystem"`	`<sub1>`	A subsystem identifier. Within the context of an expansion, be careful to escape the delimiters on a subsystem identifier as in: `%<x == <sub\>>`.
`"Unsigned"`	`15U`	A 32-bit, unsigned, integer.

Table 2-2: Target Language Values (Continued)

Value Type String	Example	Description
`"Unsigned Gaussian"`	`3+5i`	A 32-bit, unsigned, integer imaginary number.
`"Vector"`	`[ 1, 2] OR Vector(2) [1,2]`	Vectors are lists of values. The individual elements of a vector do not need to be the same type, and may be any type except vectors or matrices.

Target Language Expressions

In any place throughout a target file, you can include an expression of the form `%<expression>`. The Target Language Compiler replaces `expression` with a calculated replacement value based upon 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.

For Table 2-3, note that “numeric” is one of the following:

- Boolean
- Number
- Unsigned
- Real
- Real32
- Complex
- Complex32
- Gaussian
- UnsignedGaussian

Also, note that “integral” is one of the following:

- Number
- Unsigned
- Boolean

See “Usual Promotions” for information on the promotions that result when the Target Language Compiler operates on mixed types of expressions.

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 2-40.
`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 appends the scalar to the vector. Matrices - If the first argument is a matrix and the second is a vector of the same column-width as the matrix, it appends the vector as another row in the matrix. Records - If the first argument is a record, it adds the second argument as a parameter identifier (with its current value).

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.

Usual Promotions

When the Target Language Compiler operates on mixed types of expressions, it promotes the result to the common types indicated in Table 2-4.

This table uses the following abbreviations:

- B: Boolean
- N: Number
- U: Unsigned
- F: Real32
- D: Real
- G: Gaussian
- UG: UnsignedGaussian
- C32: Complex32
- C: Complex

The top row (in bold) and first column (in bold) show the types of expression used in the operation. The intersection of the row and column shows the resulting type of expression.

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

Table 2-4: Usual Promotions

	B	**N**	**U**	**F**	**D**	**G**	**UG**	**C32**	**C**
B	B	N	U	F	D	G	UG	C32	C
N	N	N	U	F	D	G	UG	C32	C
U	U	U	U	F	D	UG	UG	C32	C
F	F	F	F	F	D	C32	C32	C32	C
D	D	D	D	D	D	C	C	C	C
G	G	G	UG	C32	C	G	UG	C32	C
UG	UG	UG	UG	C32	C	UG	UG	C32	C
C32	C32	C32	C32	C32	C	C32	C32	C32	C
C	C	C	C	C	C	C	C	C	C

Formatting

By default, the Target Language Compiler outputs all floating-point numbers in exponential notation with 16 digits of precision. To override the default, use the directive:

```
%realformat string
```

If `string` is `"EXPONENTIAL"`, the standard exponential notation with 16 digits of precision is used. If `string` is `"CONCISE"`, the Compiler uses a set of internal heuristics to output the values in a more readable form while maintaining accuracy. The `%realformat` directive sets the default format for `Real` number output to the selected style for the remainder of processing or until it encounters another `%realformat` directive.

Conditional Inclusion

The conditional inclusion directives are

```
%if constant-expression
%else
%elseif constant-expression
%endif
```

and

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

%if

The `constant-expression` must evaluate to an integral expression. It controls the inclusion of all the following lines until it encounters a `%else`, `%elseif`, or `%endif` directive. If the `constant-expression` evaluates to `0`, the lines following the directive are not included. If the `constant-expression` evaluates to any other integral value, the lines following the `%if` directive are included up until the `%endif`, `%elseif`, or `%else` directives.

When the Compiler encounters an `%elseif` directive, and no prior `%if` or `%elseif` directive has evaluated to nonzero, the Compiler evaluates the expression. If the value is 0, the lines following the `%elseif` directive are not included. If the value is nonzero, the lines following the `%elseif` directive are included up until the subsequent `%else`, `%elseif`, or `%endif` directive.

The `%else` directive begins the inclusion of source text if all of the previous `%elseif` statements or the original `%if` statement evaluates to `0`; otherwise, it prevents the inclusion of subsequent lines up to and including the following `%endif`.

The `constant-expression` can contain any expression specified in 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

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

The syntax of the %for multiple inclusion directive is:

```
%for ident1 = const-exp1, const-exp2, ident2 = const-exp3
  %body
    %break
    %continue
  %endbody
%endfor
```

The first portion of the %for directive is identical to the %foreach statement in that it causes a loop to execute from 0 to N–1 times over the body of the loop. In the normal case, it includes only the lines between %body and %endbody, and the lines between the %for and %body, and ignores the lines between the %endbody and %endfor.

The %break and %continue directives act the same as they do in the %foreach directive.

const-exp2 is a Boolean expression that indicates whether the loop should be rolled. If const-exp2 is true, 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.

%roll

The syntax of the `%roll` multiple inclusion directive is:

```
%roll ident1 = roll-vector-exp, ident2 = threshold-exp, ...
              block-exp [, type-string [,exp-list] ]
  %break
  %continue
%endroll
```

This statement uses the `roll-vector-exp` to expand the body of the `%roll` statement multiple times as in the `%foreach` statement. If a range is provided in the `roll-vector-exp` and that range is larger than the `threshold-exp` expression, the loop will roll. When a loop rolls, the body of the loop is expanded once and the identifier (`ident2`) provided for the threshold expression is set to

the name of the loop control variable. If no range is larger than the specified rolling threshold, this statement is identical in all respects to the `%foreach` statement.

For example:

```
%roll Idx = [ 1 2 3:5, 6, 7:10 ], lcv = 10, ablock
%endroll
```

In this case, the body of the `%roll` statement expands 10 times as in the `%foreach` statement since there are no regions greater than or equal to 10. `Idx` counts from 1 to 10, and `lcv` is set to the null string, `""`.

When the Target Language Compiler determines that a given block will roll, it performs a `GENERATE_TYPE` function call to output the various pieces of the loop (other than the body). The default type used is `Roller`; you can override this type with a string that you specify. Any extra arguments passed on the `%roll` statement are provided as arguments to these special-purpose functions. The called function is one of these four functions.

RollHeader(block, ...). This function is called once on the first section of this roll vector that will actually roll. It should return a string that is assigned to the `lcv` within the body of the `%roll` statement.

LoopHeader(block, StartIdx, Niterations, Nrolled, ...). This function is called once for each section that will roll prior to the body of the `%roll` statement.

LoopTrailer(block, Startidx, Niterations, Nrolled, ...). This function is called once for each section that will roll after the body of the `%roll` statement.

RollTrailer(block, ...). This function is called once at the end of the `%roll` statement if any of the ranges caused loop rolling.

These functions should output any language-specific declarations, loop code, and so on as required to generate correct code for the loop. An example of a `Roller.tlc` file is:

```
%implements Roller "C"
%function RollHeader(block) Output
   {
       int i;
   %return ("i")
%endfunction

%function LoopHeader(block,StartIdx,Niterations,Nrolled) Output
   for (i = %<StartIdx>; i < %<Niterations+StartIdx>; i++)
   {
%endfunction

%function LoopTrailer(block,StartIdx,Niterations,Nrolled) Output
   }
%endfunction

%function RollTrailer(block) Output
   }
%endfunction
```

Note The Target Language Compiler function library provided with Real-Time Workshop has the capability to extract references to the Block I/O and other Real-Time Workshop-specific vectors that vastly simplify the body of the `%roll` statement. These functions include `LibBlockInputSignal`, `LibBlockOutputSignal`, `LibBlockParameter`, `LibBlockRWork`, `LibBlockIWork`, `LibBlockPWork`, and `LibDeclareRollVars`. For more details on these functions and other Simulink functions, see "Loop Rolling" in Chapter 4 and the `LibBlockMatrixParameter`, `LibBlockParameterAddr`, `LibBlockContinuousState`, and `LibBlockDiscreteState` function reference pages in Chapter 5. 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 Target Language Compiler code produces the output:

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

Object-Oriented Facility for Generating Target Code

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

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

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

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 2-45. 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)
```

To specify a relative path, the path must explicitly start with "`.`". For example,

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

When an explicit relative path is specified, the directory that is added to the Target Language Compiler search path is created by concatenating the location of the target file that contains the `%addincludepath` directive and the explicit relative path.

The Target Language Compiler searches the directories in the following order for target or include files:

1. The current directory
2. Any `%addincludepath` directives
3. Any include paths specified at the command line via `–I`

Typically, `%addincludepath` directives should be specified in your system target file. Multiple `%addincludepath` directives will add multiple paths to the Target Language Compiler search path.

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 Target Language Compiler places messages generated by `%trace` onto `stderr` if and only if you specify the verbose mode switch (`–v[1|2|3]`) to the Target Language 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 2-5 lists the built-in functions and values that are added to the list of parameters that appear in the `.rtw` file. These Target Language Compiler 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 2-5: Target Language Compiler 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 "FEVAL Function" later in this chapter 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 2-5: Target Language Compiler 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 "Object-Oriented Facility for Generating Target Code" earlier in this chapter.
`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 "Object-Oriented Facility for Generating Target Code" earlier in this chapter.

Table 2-5: Target Language Compiler 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.
`GET_COMMAND_SWITCH`	Returns the value of command-line switches.
`IDNUM(expr)`	`expr` must be a string. The result is a vector where the first element is a leading string (if any) and the second element is a number appearing at the end of the input string. For example: `IDNUM("ABC123") yields ["ABC", 123]`
`IMAG(expr)`	Returns the imaginary part of a complex number.
`INT8MAX`	`127`
`INT8MIN`	`-128`
`INT16MAX`	`32767`
`INT16MIN`	`-32768`
`INT32MAX`	`2147483647`
`INT32MIN`	`-2147483648`
`ISINF(expr)`	Returns 1 if the value of the expression is `inf`.
`ISNAN(expr)`	Returns 1 if the value of the expression is `NAN`.
`ISFINITE(expr)`	Returns 1 if the value of the expression is not +/- `inf` or `NAN`.

Table 2-5: Target Language Compiler Built-in Functions and Values (Continued)

Special Macro Name	Expansion
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.
REAL(expr)	Returns the real part of a complex number.
Roll_ITERATIONS()	Returns the number of times the current roll regions is looping or NULL if not inside a %roll construct.
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.

Table 2-5: Target Language Compiler Built-in Functions and Values (Continued)

Special Macro Name	Expansion
`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.
`STRINGOF(expr)`	Accepts a vector of ASCII values and returns a string that is constructed by treating each element as the ASCII code for a single character. Used primarily for S-Function string parameters in Real-Time Workshop.

`SYSNAME(expr)`	Looks for specially formatted strings of the form `<x>/y` and returns `x` and `y` as a 2-element string vector. This is used to resolve subsystem names in Real-Time Workshop. For example,
	`%<sysname("<sub>/Gain")>`
	returns
	`["sub","Gain"]`
	To expand a full Simulink path name, see "`LibGetBlockPath`" in Chapter 5, the function reference chapter.

Table 2-5: Target Language Compiler Built-in Functions and Values (Continued)

Special Macro Name	Expansion
`TLCFILES`	Returns a vector containing the names of all the target files included thus far in the expansion. See also `NUMTLCFILES`.
`TLC_TIME`	The date and time of compilation.
`TLC_VERSION`	The version and date of the Target Language Compiler.
`TYPE(expr)`	Evaluates `expr` and determines the result type. The result of this function is a string that corresponds to the type of the given expression. See the entries in the "Value Type String" column in Table 2-2, the Target Language Values table, for possible values.
`UINT8MAX`	`255U`
`UINT8MIN`	`65535U`
`UINT32MAX`	`4294967295U`
`WHITE_SPACE(expr)`	Accepts a string and returns 1 if the string contains only whitespace characters (, `\t`, `\h`, `\r`); returns 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.

The following table shows the conversions that are made when calling MATLAB.

TLC Type	MATLAB Type
"Boolean" or "Number" or “Real”	Double Scalar
"Real32”	Double Scalar
"Unsigned"	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.
"Scope"	MATLAB structure with elements

When values are returned from MATLAB, they are converted as shown in the following table.

MATLAB Type	TLC Type
Double Scalar	"Number" or "Real" depending on the value
Complex Scalar	"Complex"
Char Row Vector	"String"

MATLAB Type	TLC Type
`Char Matrix` or `Char Col Vector`	`"Vector"` of `"Strings"`
`Double Vector`	`"Vector"` whose elements are `"Number"`,`"Real"`, or `"Complex"`, 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 )
```

The following table shows the standard exponential notation (Constant Form) and the Target Language Compiler type.

Constant Form	TLC Type
`1.0`	`"Real"`
`1.0[F/f]`	`"Real32"`
`1`	`"Number"`
`1[U|u]`	`"Unsigned"`
1.0i	`"Complex"`
`1[Ui|ui]`	`"Unsigned Gaussian"`
`1i`	`"Gaussian"`
`1.0[Fi|fi]`	`"Complex32"`

Note The suffix used controls the Target Language Compiler type obtained from the constant.

The following table shows Target Language Compiler constants and their equivalent MATLAB values.

TLC Constant(s)	Equivalent MATLAB Value
`rtInf`, `Inf`, `inf`	`+inf`
`rtMinusInf`	`-inf`
`rtNan`, `NaN`, `nan`	`nan`
`rtInfi`, `Infi`, `infi`	`inf*i`
`rtMinusInfi`	`-inf*i`
`rtNaNi`, `NaNi`, `nani`	`nan*i`

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. Real-Time Workshop 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 Real-Time Workshop file without qualifying it with the scope. This example does not generate an error:

```
%assign CompiledModel.Name = "newname"  %% No error
```

This example generates an error:

```
%with CompiledModel
  %assign Name = "newname"              %% Error
%endwith
```

Creating Records

Use the `%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:

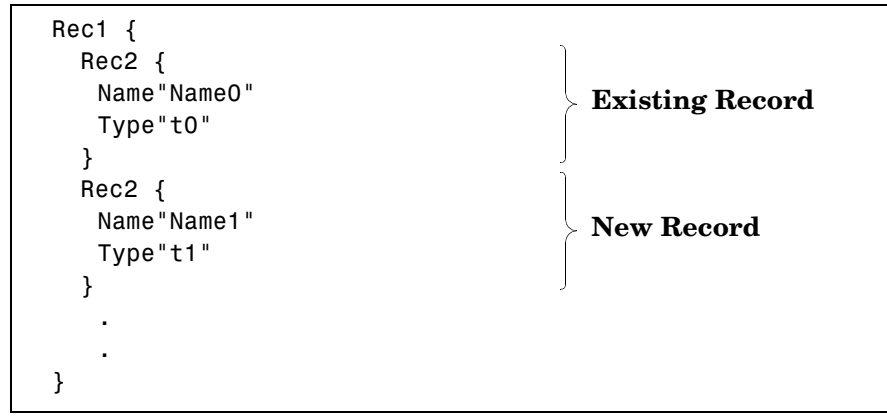


Figure 2-1: Creating a New Record

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:

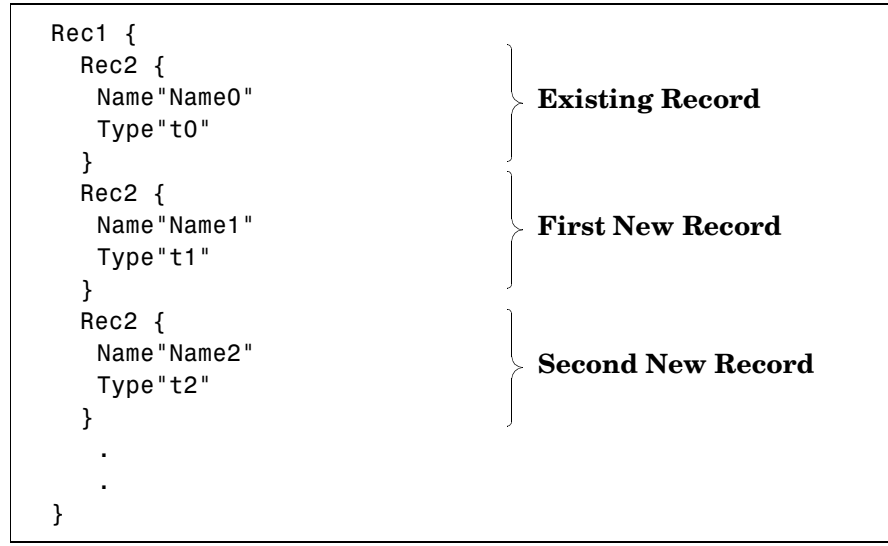


Figure 2-2: Creating Multiple Records

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.

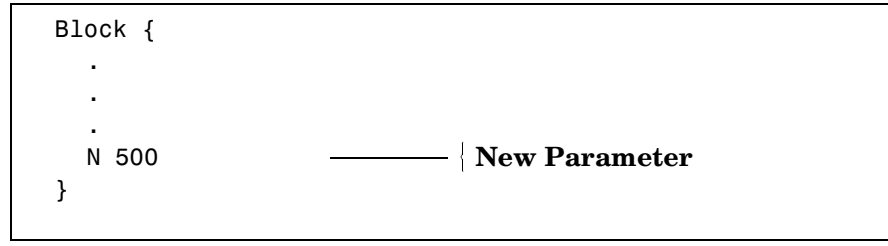


Figure 2-3: Parameter Added to Existing Record

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, if you have the following Real-Time Workshop file:

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

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

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

or by using `%with`:

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

Variable Scoping

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

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

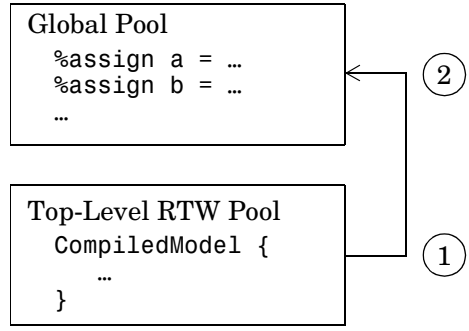


Figure 2-4: Search Sequence

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

Example

When you add the following construct

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

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

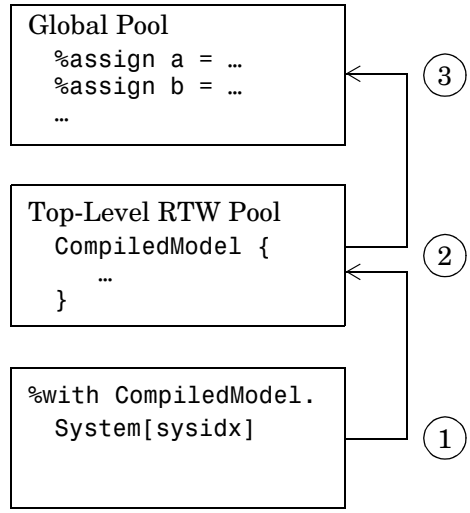


Figure 2-5: Modifying the Search Sequence

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

```
Name
```

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

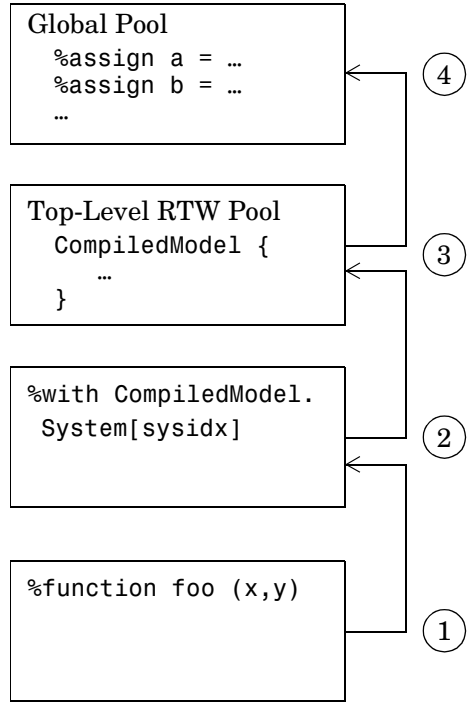


Figure 2-6: Scoping Rules Within Functions

For example, if you have the following code,

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

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

Target Language Functions

The target language function construct is:

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

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

A function optionally returns a value with the `%return` directive. The returned value can be any of the types defined in the in Table 2-2, 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:

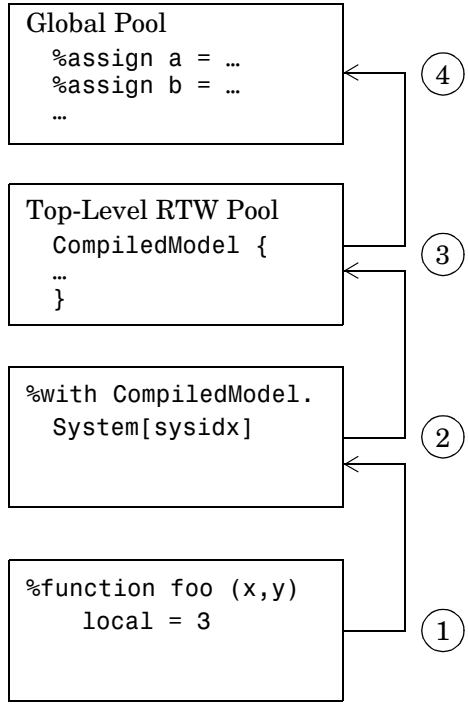


Figure 2-7: Scoping Rules Within Functions Containing Local Variables

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,

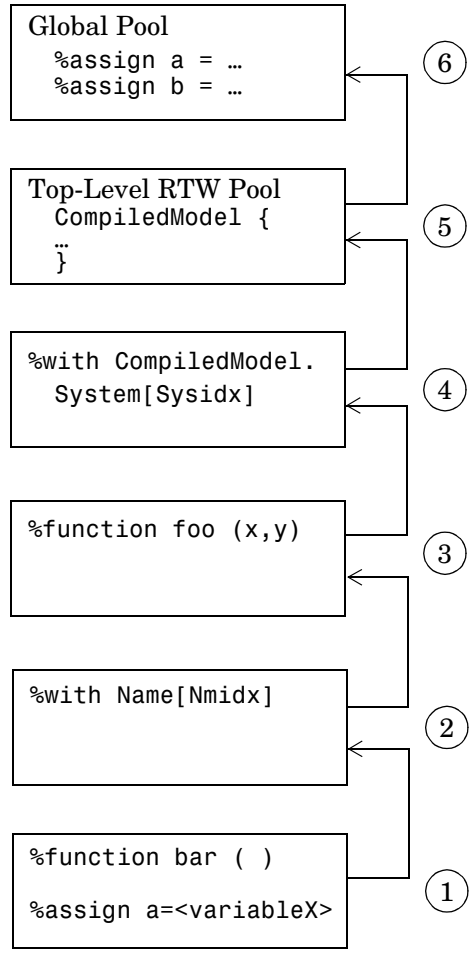


Figure 2-8: Scoping Rules When Using %with Within a Function

%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
```

Command Line Arguments

To call the Target Language Compiler, use:

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

Table 2-6 lists the switches you can use with the Target Language Compiler. Order makes no difference. Note that if you specify a switch more than once, the last one takes precedence.

Table 2-6: Target Language Compiler Switches

Switch	Meaning
`–r filename`	Reads a database file (such as a `.rtw` file). Repeat this option multiple times to load multiple database files into the Target Language Compiler. Omit this option for target language programs that do not depend on the database.
`–v[number]`	Sets the internal verbose level to `<number>`. Omitting this option sets the verbose level to `1`.
`–Ipath`	Adds the specified directory to the list of paths to be searched for TLC files.
`–Opath`	Specifies that all output produced should be placed in the designated directory, including files opened with `%openfile` and `%closefile`, and `.log` files created in debug mode. To place files in the current directory, use `–O` (use the capital letter O, not zero).
`–m[number]`	Specifies the maximum number of errors to report is `<number>`. If no `–m` argument appears on the command line, it defaults to reporting the first five errors. If the `<number>` argument is omitted on this option, 1 is assumed.

Table 2-6: 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, <expr>, for the identifier, <ident>, for some parameters; equivalent to the %assign command.

As an example, the command line

```
tlc –r Demo.rtw –v grt.tlc
```

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

Filenames and Search Paths

All target files have the .tlc extension. By default, block-level files have the same name as the Type of the block in which they appear. You can override the search path for target files with your own local versions. The Target Language Compiler finds all target files along this path. If you specify additional search paths with the –I switch of the tlc command or via the %addincludepath directive, they will be searched after the current working directory, and in the order in which you specify them.

Target Language Debug Mode

When you initiate the debug mode via the –d switch of the tlc command, the Target Language 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

Target Language Files

Introduction

This chapter describes target files included with the Target Language Compiler. You must write or modify a target file if you need to do one of the following:

- Customize the code generated for a block

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

- Inline an S-function

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

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

- Customize the code generated for all models

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

- Implement support for a new language

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

The first section of this chapter describes the architecture of the Real-Time Workshop from the standpoint of the Target Language Compiler and target files. The second section describes block target files and how to write them. The final section discusses recommended coding conventions.

Refer to Chapter 2, “Working with the Target Language,” for a description of the Target Language and Chapter 4, “Writing Target Language Files: A Tutorial,” for a tutorial on using the Target Language and a description of how to inline S-functions.

Real-Time Workshop and Target Language Compiler Architecture

The Target Language Compiler works with Simulink to generate code. This section describes in detail the architecture shown in the sections of Figure 1-1 marked *Target Files* and *Target Language Compiler*.

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

Target Files

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

- System target files

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

- Block target files

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

System Target Files

The entire code generation process starts with the single system target file that you specify in the Real-Time Workshop page of the **Simulation Parameters** dialog box. A close examination of a system target file reveals how code generation occurs. This a listing of the non-comment lines in `grt.tlc`, the target file to generate code for a generic real-time executable.

```
%selectfile NULL_FILE

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

%include "codegenentry.tlc"
```

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

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

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

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

%include "commonsetup.tlc"

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

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

Generated code is placed in a model or subsystem function. The relevant generated function names and their execution order is detailed in Chapter 6 of the *Real-Time Workshop User's Guide*. During code generation, functions from each of the block target files are executed and the generated code is placed in the appropriate model or subsystem functions.

Block Target Files

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

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

Block Target File Mapping

The *block target file mapping* specifies which target file should be used to generate code for which block type. This mapping resides in *matlabroot*`/rtw/c/tlc/genmap.tlc` and is also reproduced in Appendix A. All the TLC files listed are located in *matlabroot*`/rtw/c/tlc` for C and *matlabroot*`/rtw/ada/tlc` for Ada.

Block Functions

The functions declared inside each of the block target files are called by the system target files. In these tables, `block` refers to a Simulink block name (e.g., `gain` for the Gain block) and `system` refers to the subsystem in which the block resides. The first table lists the two functions that are used for preprocessing and setup. Neither of these functions outputs any generated code.

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

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

`BlockInstanceData(block, system)`
`Enable(block, system)`
`Disable(block, system)`
`Start(block, system)`

`InitializeConditions(block, system)`
`Outputs(block, system)`
`Update(block, system)`
`Derivatives(block, system)`
`Terminate(block, system)`

In object-oriented programming terms, these functions are polymorphic in nature since each block target file contains the same functions. The Target Language Compiler dynamically determines at run-time which block function to execute depending on the block's type. That is, the system file only specifies that the `Outputs` function, for example, is to be executed. The particular `Outputs` function is determined by the Target Language Compiler depending on the block's type.

To write a block target file, use these polymorphic block functions combined with the Target Language Compiler library functions. For a complete list of the Target Language Compiler library functions, see Chapter 5, "Target Language Compiler Function Library Reference."

To inline an S-function, see Chapter 4, "Writing Target Language Files: A Tutorial." The end of the section on inlining S-functions also lists useful functions classified by purpose.

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 Chapter 5, "Target Language Compiler Function Library Reference," for a list of relevant functions to call from inside this block function. See `lookup2d.tlc` for an example of the `BlockInstanceSetup` function.

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

This example uses `BlockInstanceSetup`.

```
%function BlockInstanceSetup(block, system) void
%if (block.InMask == "yes")
    %assign blockName = LibParentMaskBlockName(block)
  %else
    %assign blockName = LibGetFormattedBlockPath(block)
  %endif
  %if (CodeFormat == “Embedded-C”) || (CodeFormat == "Ada")
    %if !(ParamSettings.ColZeroTechnique == "NormalInterp" && ...
          ParamSettings.RowZeroTechnique == "NormalInterp")
      %selectfile STDOUT

Note: Removing repeated zero values from the X and Y axes will
produce more efficient code for block: %<blockName>.  To locate
this block, type

open_system('%<blockName>')

at the MATLAB command prompt.

      %selectfile NULL_FILE
    %endif
  %endif

%endfunction
```

BlockTypeSetup(block, system)

`BlockTypeSetup` executes once per block type before code generation begins. That is, if there are 10 Lookup Table blocks in the model, the `BlockTypeSetup` function in `look_up.tlc` is only called one time. Use this function to perform general work for all blocks of a given type.

See Chapter 5, “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 the S-function `foo` requiring a `#define` and two function declarations in the header file, you could define the following function.

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

  %% Place a #define in the model's header file

  %openfile buffer
    #define A2D_CHANNEL 0
  %closefile buffer

  %<LibCacheDefine(buffer)>

  %% Place function prototypes in the model's header file

  %openfile buffer
    void start_a2d(void);
    void reset_a2d(void);
  %closefile buffer

  %<LibCacheFunctionPrototype(buffer)>

%endfunction
```

The remaining block functions execute once for each block in the model.

BlockInstanceData(block, system)

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

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

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

```
%function BlockInstanceData(block, system) Output
  %assign pwork = LibBlockPWork(SemID, "", "", 0)
  %if EXISTS("ssBlock")
    /* %<Type> Block: %<Name> */
    %% Declare the semaphore depending on the Code Format.
    /*      VxWorks binary semaphore for task: %<ssBlock.Name> */
    %if UsingMalloc == 1
      %<pwork> = malloc(sizeof(SEM_ID));
      rt_VALIDATE_MEMORY(%<tSimStruct>, %<pwork>);
    %else
      static SEM_ID %<taskSemaphoreName>;
      %<pwork> = (void *)&%<taskSemaphoreName>;
    %endif
  %endif
%endfunction

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

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

Enable(block, system)

Nonvirtual subsystem `Enable` functions are created whenever a Simulink subsystem contains a block with an `Enable` function. Including the `Enable`

function in a block's target file places the block's specific enable code into this subsystem `Enable` function. See `sin_wave.tlc` for an example of the `Enable` function.

```
%% Function: Enable ============================================
%% Abstract:
%% Subsystem Enable code is only required for the discrete form
%% of the Sine Block. Setting the boolean to TRUE causes the
%% Output function to re-sync its last values of cos(wt) and
%% sin(wt).
%%
%function Enable(block, system) Output
  %if LibIsDiscrete(TID)
    /* %<Type> Block: %<Name> */
    %<LibBlockIWork(SystemEnable, "", "", 0)> = (int_T) TRUE;

  %endif
%endfunction
```

Disable(block, system)

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

Start(block, system)

Include a `Start` function to place code into the start function. The code inside the `Start` function executes once and only once. Typically, you include a `Start` function to execute code once at the beginning of the simulation (e.g., initialize values in the work vectors; see `backlash.tlc`) or code that does not need to be

re-executed when the subsystem in which it resides enables. See `constant.tlc` for an example of the `Start` function.

```
%% Function: Start ============================================
%% Abstract:
%% Set the output to the constant parameter value if the block
%% output is visible in the model's start function scope, i.e.,
%% it is in the global rtB structure.
%%
%function Start(block, system) Output
  %if  LibBlockOutputSignalIsInBlockIO(0)
    /* %<Type> Block: %<Name> */
    %assign rollVars = ["Y", "P"]
    %roll idx = RollRegions, lcv = RollThreshold, block, ...
      "Roller", rollVars
      %assign yr = LibBlockOutputSignal(0,"", lcv, ...
        "%<tRealPart>%<idx>")
      %assign pr = LibBlockParameter(Value, "", lcv, ...
        "%<tRealPart>%<idx>")
      %<yr> = %<pr>;
      %if LibBlockOutputSignalIsComplex(0)
        %assign yi = LibBlockOutputSignal(0, "", lcv, ...
          "%<tImagPart>%<idx>")
        %assign pi = LibBlockParameter(Value, "", lcv, ...
          "%<tImagPart>%<idx>")
        %<yi> = %<pi>;
      %endif
    %endroll

  %endif
%endfunction %% Start
```

InitializeConditions(block, system)

TLC code that is generated from the block's `InitializeConditions` function ends up in one of two places. A nonvirtual subsystem contains an `Initialize` function when it is configured to reset states on enable. In this case, the TLC code generated by this block function is placed in the subsystem `Initialize` function and the `start` function will call this subsystem `Initialize` function. If, however, the Simulink block resides in the root system or in a nonvirtual

subsystem that does not require an `Initialize` function, the code generated from this block function is placed directly (inlined) into the `start` function.

There is a subtle difference between the block functions `Start` and `InitializeConditions`. Typically, you include a `Start` function to execute code that does not need to re-execute when the subsystem in which it resides enables. You include an `InitializeConditions` function to execute code that must re-execute when the subsystem in which it resides enables. See `delay.tlc` for an example of the `InitializeConditions` function. The following code is an example from `ratelim.tlc`:

```
%% Function: InitializeConditions ============================
%%
%% Abstract:
%% Invalidate the stored output and input in rwork[1 ...
%% 2*blockWidth] by setting the time stamp (stored in
%% rwork[0]) to rtInf.
%%
%function InitializeConditions(block, system) Output
  /* %<Type> Block: %<Name> */
  %<LibBlockRWork(PrevT, "", "", 0)> = %<LibRealNonFinite(inf)>;

%endfunction %% InitializeConditions
```

Outputs(block, system)

A block should generally include an `Outputs` function. The TLC code generated by a block's `Outputs` function is placed in one of two places. The code is placed directly in the model's `Outputs` function if the block does not reside in a nonvirtual subsystem and in a subsystem's `Outputs` function if the block

resides in a nonvirtual subsystem. See `absval.tlc` for an example of the `Outputs` function:

```
%% Function: Outputs ========================================
%% Abstract:
%%      Y[i] = fabs(U[i]) if U[i] is real or
%%      Y[i] = sqrt(U[i].re^2 + U[i].im^2) if U[i] is complex.
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %%
  %assign inputIsComplex = LibBlockInputSignalIsComplex(0)
  %assign RT_SQUARE = "RT_SQUARE"
  %%
  %assign rollVars = ["U", "Y"]
  %if inputIsComplex
    %roll sigIdx = RollRegions, lcv = RollThreshold, ...
      block, "Roller", rollVars
      %%
      %assign ur = LibBlockInputSignal( 0, "", lcv, ...
        "%<tRealPart>%<sigIdx>")
      %assign ui = LibBlockInputSignal( 0, "", lcv, ...
        "%<tImagPart>%<sigIdx>")
      %%
      %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
     %<y> = sqrt( %<RT_SQUARE>( %<ur> ) + %<RT_SQUARE>( %<ui> ) );
    %endroll
  %else
    %roll sigIdx = RollRegions, lcv = RollThreshold, ...
      block, "Roller", rollVars
      %assign u = LibBlockInputSignal (0, "", lcv, sigIdx)
      %assign y = LibBlockOutputSignal(0, "", lcv, sigIdx)
      %<y> = fabs(%<u>);
    %endroll
  %endif

%endfunction
```

Note Zero-crossing reset code is placed in the `Outputs` function.

Update(block, system)

Include an `Update` function if the block has code that needs to be updated at each major time step. Code generated from this function is either placed into the model's or the subsystem's `Update` function, depending on whether or not the block resides in a nonvirtual subsystem. See `delay.tlc` for an example of the `Update` function:

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

%endfunction %% Update
```

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

Derivatives(block, system)

Include a `Derivatives` function when generating code to compute the block's continuous states. Code generated from this function is either placed into the `model's` or the subsystem's `Derivatives` function, depending on whether or not the block resides in a nonvirtual subsystem. See `integrat.tlc` for an example of the `Derivatives` function.

Terminate(block, system)

Include a `Terminate` function to place any code into `MdlTerminate`. User-defined S-function target files can use this function to save data, free memory, reset hardware on the target, and so on. See `tofile.tlc` for an example of the `Terminate` function.

Coding Conventions

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

1 All identifiers in the Real-Time Workshop file begin with a capital letter. For example,

```
NumContStates     10
NumBlocks         52
```

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 may contain

```
Block {
  :
  TID0
  NumRWorkDefines 1
  RWorkDefine {
    Name  "PrevT"
    Width  1
  }
  PrevTRWorkDefine[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 Target Language Compiler assignment directive (`%assign`) generates a warning if you assign a value to an "unqualified" Real-Time Workshop identifier. For example,

```
%assign TID = 1
```

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 Real-Time Workshop variables. An example of a global TLC variable defined in `mdlwide.tlc` is

   ```
   %assign InlineParameters = 1
   ```

   An example of a global reference in a function is

   ```
   %function foo() void
     %assign ::GlobalIdx = ::GlobalIdx + 1
   %endfunction
   ```

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. 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.

---

5. Do not hard code the variables defined in `commonsetup.tlc`. Since the Real-Time Workshop tracks use of variables and generates code based on usage, you should use access routines instead of directly using a variable. For example, you should not use the following in your TLC file:

   ```
   x = %<tInf>;
   ```

   Instead, use

   ```
   x = %<LibRealNonFinite(inf)>;
   ```

Similarly, instead of using `%<tTID>`, use `%<LibTID()>`. For a complete list of functions, see Chapter 5, "Target Language Compiler Function Library Reference."

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

Avoid naming global variables in your run-time interface modules that start with `rt` or `rt_` since they may conflict with Real-Time Workshop global variables and functions. These TLC variables are declared in `commonsetup.tlc`.

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

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

Notes about this TLC code:

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

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

c Try to stay within 80 columns per line for the function banner. You might set up an 80 column comment line at the top of each function. As an example, see `constant.tlc`.

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

e Use the variable `rollVars` when using the `%roll` construct.

f When naming loop control variables, use `sigIdx` and `lcv` when looping over `RollRegions` and `xidx` and `xlcv` when looping over the states.

Example: `Output` function in `gain.tlc`

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

Example: `InitializeConditions` function in `linblock.tlc`

```
%roll xidx = [0:nStates-1], xlcv = RollThreshold,...
       block, "Roller", rollVars
```

6 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.

Built-In Target Language Compiler Functions

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

STRINGOF(value)

Given an Real-Time Workshop string vector, this built-in 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 *matlabroot*`/rtw/c/mwdspace/devices/dp_read.tlc`.

EXISTS("name")

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

SIZE(value, n)

The behavior of this built-in function is shown below.

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

4

Writing Target Language Files: A Tutorial

Introduction

This tutorial chapter describes how to write target language files. It describes the target language as well as the Target Language Compiler and includes several simple examples. It also explains how to inline an S-function and provides an example. It concludes with a discussion of loop rolling.

The Target Language

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

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

Output Streams

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

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

To run this Target Language Compiler program, type

```
tlc hello.tlc
```

at the MATLAB prompt.

This simple program demonstrates some important concepts underlying the purpose (and hence the design) of the Target Language Compiler. Since the primary purpose of the Target Language Compiler is to generate code, it is output (or stream) oriented. It makes it easy to handle buffers of text and output them easily. In the above program, the `%selectfile` directive tells the Target Language Compiler to send any following text that it does not recognize to the standard output device. All syntax that the Target Language Compiler recognizes begins with the `%` character. Since `Hello, World` is not recognized,

it is sent directly to the output. You could just as easily change the output destination to be a file:

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

Note that you can switch between buffers to display status messages. The semantics of the three directives, `%openfile`, `%selectfile`, and `%closefile` are given in Chapter 2.

Variable Types

The absence of explicit type declarations for variables is another feature of the Target Language Compiler. See Chapter 2, "Working with the Target Language," for more information on the implicit data types of variables.

Records

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

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

where `recVar` is the name of the variable that references this record while `recType` is the record itself. `fieldi` is a string and `valuei` is the corresponding Target Language Compiler value.

Records can have nested records, or subrecords, within them. The *`model`*`.rtw` file is essentially one large record, named `CompiledModel`, containing several subrecords. Thus, a simple program that loops through a model and outputs the name of all blocks in the model would look like the following code.

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

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

Like Pascal, the Target Language Compiler provides a `%with` directive that facilitates using records. See Chapter 2 for a detailed description of the directive and its associated scoping rules.

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

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

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

Functions such as `LibGetFormattedBlockPath` are provided in the Target Language Compiler libraries located in *`matlabroot`*`/rtw/c/tlc/*lib.tlc`. For a complete list of available functions, refer to Chapter 5, “Target Language Compiler Function Library Reference.”

The Target Language Compiler

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

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

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

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

A Basic Example

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

Process

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

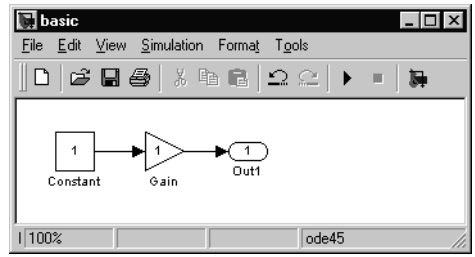


Figure 4-1: Simulink Model

Selecting **Parameters** from Simulink's **Simulation** menu displays the **Simulation Parameters** dialog box shown in Figure 4-2.

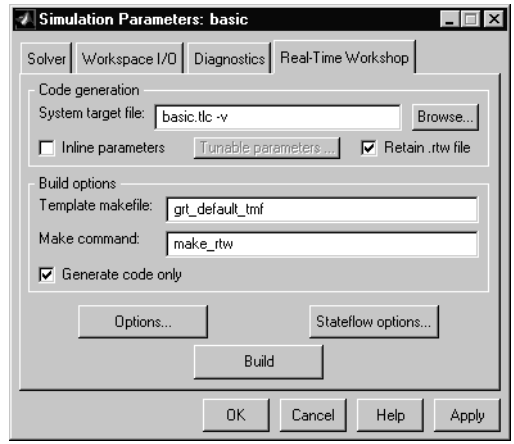


Figure 4-2: Simulation Parameters Dialog

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

```
%with CompiledModel

My model's name is: %<Name>

It was generated on: %<GeneratedOn>

And it has %<NumContStates> continuous states.
%endwith
```

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               "3.0 $Date: 1998/12/18 22:48:41 $"
  ModelVersion          "1.1"
  GeneratedOn           "Mon Jan 04 18:42:50 1999"
  Solver                FixedStepDiscrete
  SolverType            FixedStep
  StartTime             0.0
  StopTime              10.0
  FixedStepOpts {
    SolverMode            SingleTasking
    FixedStep             0.2
       .
       .
       .
  NumModelInputs            0
  NumModelOutputs           1
  NumNonVirtBlocksInModel   0
  DirectFeedthrough         no
  NumContStates             0
  NumDiscStates             0
       .
       .
       .
```

```
  BlockOutputs {
    BlockOutputDefaults {
      TestPoint                no
        .
        .
        .
System {
    Type                     root
    Name                     "<Root>"
    Identifier               root
    NumBlocks                3
        .
        .
        .
}
```

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: Mon Jan 04 18:42:50 1999

And it has 0 continuous states.
```

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

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 Application Program Interface (API) layer from the generated code.

Level 1 C MEX 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:

Function	Purpose
`mdlInitializeSizes`	Initialize the sizes array.
`mdlInitializeSampleTimes`	Initialize the sample times array.
`mdlInitializeConditions`	Initialize the states.
`mdlOutputs`	Compute the outputs.
`mdlUpdate`	Update discrete states.
`mdlDerivatives`	Compute the derivatives of continuous states.
`mdlTerminate`	Clean up when the simulation terminates.

Level 2 C MEX S-functions that are not inlined make calls to the above functions with the following exceptions:

`mdlInitializeConditions` is only called if `MDL_INITIALIZE_CONDITIONS` is declared with `#define`.

`mdlStart` is called only if `MDL_START` is declared with `#define`.

`mdlUpdate` is called only if `MDL_UPDATE` is declared with `#define`.

`mdlDerivatives` is called only if `MDL_DERIVATIVES` is declared with `#define`.

By inlining an S-function, you can eliminate the calls to these possibly empty functions in the simulation loop. This can greatly improve the efficiency of the generated code. To inline an S-function called `sfunc_name`, you create a custom S-function block target file called `sfunc_name.tlc` and place it in the same directory as the S-function's MEX-file. Then, at build time, the target file is

executed instead of setting up function calls into the S-function's `.c` file. The S-function target file *inlines* the S-function by directing the Target Language Compiler to insert only the statements defined in the target file.

In general, inlining an S-function is especially useful when:

- The time required to execute the contents of the S-function is small in comparison to the overhead required to call the S-function.
- Certain S-function routines are empty (e.g., `mdlUpdate`).
- The behavior of the S-function changes between simulation and code generation. For example, device driver I/O S-functions may read from the MATLAB workspace during simulation, but read from an actual hardware address in the generated code.

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 Level 2 S-function is `foogain`, the C MEX S--function must contain this code.

```
#define S_FUNCTION_NAME foogain
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"
#define GAIN mxGetPr(ssGetSFcnParam(S,0))[0]

static void mdlInitializeSizes(SimStruct *S)
{
  ssSetNumContStates     (S, 0);
  ssSetNumDiscStates     (S, 0);

  if (!ssSetNumInputPorts(S, 1)) return;
  ssSetInputPortWidth            (S, 0, 1);
  ssSetInputPortDirectFeedThrough(S, 0, 1);

  if (!ssSetNumOutputPorts(S, 1)) return;
  ssSetOutputPortWidth          (S, 0, 1);

  ssSetNumSFcnParams     (S, 1);
  ssSetNumSampleTimes    (S, 0);
  ssSetNumIWork          (S, 0);
  ssSetNumRWork          (S, 0);
  ssSetNumPWork          (S, 0);
}

static void
mdlOutputs(SimStruct *S, int_T tid)
{
  real_T *y = ssGetOutputPortRealSignal(S, 0);
  const InputRealPtrsType u = ssGetInputPortRealSignalPtrs(S, 0);

  y[0] = (*u)[0] * GAIN;
}

static void
mdlInitializeSampleTimes(SimStruct *S){}
```

```
static void
mdlTerminate(SimStruct *S) {}

#define MDL_RTW  /* Change to #undef to remove function */
#if defined(MDL_RTW)&&(defined(MATLAB_MEX_FILE)||defined(NRT))
static void
mdlRTW (SimStruct *S)
{
  if (!ssWriteRTWParameters(S, 1,SSWRITE_VALUE_VECT,"Gain","",
                            mxGetPr(ssGetSFcnParam(S,0)),1))
  {
    return;
  }
}
#endif

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

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

For information about how to generate code with the Real-Time Worskhop, see the *Real-Time Workshop User's Guide*.

Comparison of Noninlined and Inlined Versions of model.c

Without a TLC file to define the S-function specifics, the Real-Time Workshop must call the MEX-file S-function through the S-function API. The code below

is the *model*.c file for the noninlined S-function (i.e., no corresponding TLC file).

```
/*
 * model.c
.
.
.
*/
real_T untitled_RGND = 0.0;             /* real_T ground */
/* Start the model */
void MdlStart(void)
{
  /* (no start code required) */
}
/* Compute block outputs */
void MdlOutputs(int_T tid)
{
  /* Level2 S-Function Block: <Root>/S-Function (foogain) */
  {
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnOutputs(rts, tid);
  }
}
/* Perform model update */
void MdlUpdate(int_T tid)
{
  /* (no update code required) */
}
/* Terminate function */
void MdlTerminate(void)
{
  /* Level2 S-Function Block: <Root>/S-Function (foogain) */
  {
    SimStruct *rts = ssGetSFunction(rtS, 0);
    sfcnTerminate(rts);
  }
}
#include "model.reg"
/* [EOF] model.c */
```

These are calls to the S-function API. Also, notice that `SimStruct` memory is allocated for the S-function.

These are calls to the S-function API.

This code is *model*.c with the foogain S-function fully inlined.

```
/*
 * model.c
.
.
.
*/
/* Start the model */
void MdlStart(void)
{
  /* (no start code required) */
}

/* Compute block outputs */
void MdlOutputs(int_T tid)

  /* S-Function block: <Root>/S-Function */
  rtB.S_Function = 0.0 * rtP.S_Function_Gain;
}

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

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

#include "model.reg"

/* [EOF] model.c */
```

There are no calls to the S-function API in the inlined version of *model*.c. Also, note that there is no child SimStruct for the S-function.

Inlining eliminates any unnecessary calls to S-function API.

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

Including a TLC file drastically decreased the code size and increased the execution efficiency of the generated code. These notes highlight some information about the TLC code and the generated output:

- The TLC directive `%implements` is required by all block target files, and must be the first executable statement in the block target file. This directive guarantees that the Target Language Compiler does not execute an inappropriate target file for S-function `foogain`.
- The input to `foo` is set to 0 since `foo` is the only block in the model, and its input is unconnected.
- Including a TLC file for `foogain` eliminated the need for an S-function registration segment for `foogain`. This significantly reduces code size.
- The TLC code will inline the gain parameter when Real-Time Workshop is configured to inline parameter values. For example, if the S-function parameter is specified as 2.5 in the S-function dialog box, the TLC `Outputs` function generates

  ```
  rtB.foo = input * 2.5;
  ```

- Use the `%generatefile` directive if your operating system has 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`.

Comparison of Noninlined and Inlined Versions of model.reg

Inlining a Level 2 S-function significantly reduces the size of the *model*`.reg` code. Model registration functions are lengthy; much of the code has been eliminated in this example. The code below highlights the difference between

the noninlined and inlined versions of `model.reg`; inlining eliminates all this code.

```
/*
 * model.reg
 *
.
.
.
*/
/* Normal model initialization code independent of
      S-functions  */


  /* child S-Function registration */
  ssSetNumSFunctions(rtS, 1);

  /* register each child */
  {
    static SimStruct childSFunctions[1];
    static SimStruct *childSFunctionPtrs[1];

    (void)memset((char_T *)&childSFunctions[0], 0,
                 sizeof(childSFunctions));
    ssSetSFunctions(rtS, &childSFunctionPtrs[0]);
    {
      int_T i;

      for(i = 0; i < 1; i++) {
        ssSetSFunction(rtS, i, &childSFunctions[i]);
      }
    }

    /* Level2 S-Function Block: untitled/<Root>/S-Function
       (foogain) */
    {
      extern void foogain(SimStruct *rts);
      SimStruct *rts = ssGetSFunction(rtS, 0);

      /* timing info */
```

```
static time_T sfcnPeriod[1];
static time_T sfcnOffset[1];
static int_T sfcnTsMap[1];

{
  int_T i;

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

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

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

  /* port 0 */
  {
    static real_T const *sfcnUPtrs[1];

    sfcnUPtrs[0] = &untitled_RGND;
    ssSetInputPortWidth(rts, 0, 1);
    ssSetInputPortSignalPtrs(rts, 0,
                             (InputPtrsType)&sfcnUPtrs[0]);
  }
}

/* outputs */
{
  static struct _ssPortOutputs outputPortInfo[1];
  _ssSetNumOutputPorts(rts, 1);
  ssSetPortInfoForOutputs(rts, &outputPortInfo[0]);
  ssSetOutputPortWidth(rts, 0, 1);
```

```
      ssSetOutputPortSignal(rts, 0, &rtB.S_Function);
    }

    /* path info */
    ssSetModelName(rts, "S-Function");
    ssSetPath(rts, "untitled/S-Function");
    ssSetParentSS(rts, rtS);
    ssSetRootSS(rts, ssGetRootSS(rtS));
    ssSetVersion(rts, SIMSTRUCT_VERSION_LEVEL2);

    /* parameters */
    {
      static mxArray const *sfcnParams[1];

      ssSetSFcnParamsCount(rts, 1);
      ssSetSFcnParamsPtr(rts, &sfcnParams[0]);

      ssSetSFcnParam(rts, 0, &rtP.S_Function_P1Size[0]);
    }

    /* registration */
    foogain(rts);

    sfcnInitializeSizes(rts);
    sfcnInitializeSampleTimes(rts);

    /* adjust sample time */
    ssSetSampleTime(rts, 0, 0.2);
    ssSetOffsetTime(rts, 0, 0.0);
    sfcnTsMap[0] = 0;

   /* Update the InputPortReusable and BufferDstPort flags for
      each input port */
    ssSetInputPortReusable(rts, 0, 0);
    ssSetInputPortBufferDstPort(rts, 0, -1);

   /* Update the OutputPortReusable flag of each output port */
  }
}
```

TLC File to Inline S-Function foogain

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

```
%implements "foogain" "C"

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

%endfunction
```

Useful Library Functions Grouped by Purpose

The following lists group TLC library functions by purpose.

Input Signals to a Block:

- `LibBlockInputSignal`
- `LibBlockInputSignalAddr`
- `LibBlockInputSignalBufferDstPort`
- `LibBlockInputSignalWidth`

Output Signals from a Block:

- `LibBlockOutputSignal`
- `LibBlockOutputSignalAddr`
- `LibBlockOutputSignalIsInBlockIO`
- `LibBlockOutputSignalWidth`

Sources Signals of a Block:

- `LibBlockSrcSignalBlock`
- `LibBlockSrcSignalIsDiscrete`

- LibBlockSrcSignalIsGlobalAndModifiable
- LibBlockSrcSignalIsInvariant

Parameters of a Block:

- LibBlockParameter
- LibBlockParameterAddr
- LibBlockParameterSize
- LibBlockParameterValue
- LibBlockParameterFormattedValue
- LibBlockMatrixParameter
- LibBlockMatrixParameterAddr
- LibBlockMatrixParameterValue
- LibBlockMatrixParameterFormattedValue

Other Block-Related Functions:

- LibBlockContinuousState
- LibBlockDiscreteState
- LibBlockIWork
- LibBlockPWork
- LibBlockRWork
- LibBlockMode
- LibBlockSampleTime
- LibGetBlockPath
- LibGetFormattedBlockPath

Functions to Add Custom Code to the Generated Code:

- LibCacheDefine
- LibCacheExtern
- LibCacheFunctionPrototype
- LibCacheIncludes
- LibCacheNonFiniteAssignment
- LibCacheTypedefs
- LibHeaderFileCustomCode
- LibPrmFileCustomCode
- LibRegFileCustomCode

- LibSourceFileCustomCode
- LibMdlStartCustomCode
- LibMdlTerminateCustomCode
- LibMdlRegCustomCode
- LibSystemInitializeCustomCode
- LibSystemOutputCustomCode
- LibSystemUpdateCustomCode
- LibSystemDerivativeCustomCode
- LibSystemEnableCustomCode
- LibSystemDisableCustomCode
- LibSystemUserCodeIsEmpty

Utility Functions Related to Sample Times:

- LibIsContinuous
- LibIsDiscrete
- LibIsFirstInitCond
- LibIsSampleHit
- LibIsSingleRateSystem
- LibIsSpecialSampleHit
- LibGetTaskTimeFromTID
- LibGetGlobalTIDFromLocalSFcnTID
- LibGetNumSFcnSampleTimes
- LibIsSFcnSampleHit
- LibIsSFcnSingleRate
- LibIsSFcnSpecialSampleHit
- LibGetSFcnTIDType
- LibTID

Miscellaneous Utility Functions:

- LibCallFCSS
- LibAddIdentifier
- LibAddToCompiledModel
- LibGetT
- LibIsEmpty
- LibIsEqual
- LibIsFinite

- `LibIndexStruct`
- `LibRealNonFinite`

Loop Rolling

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

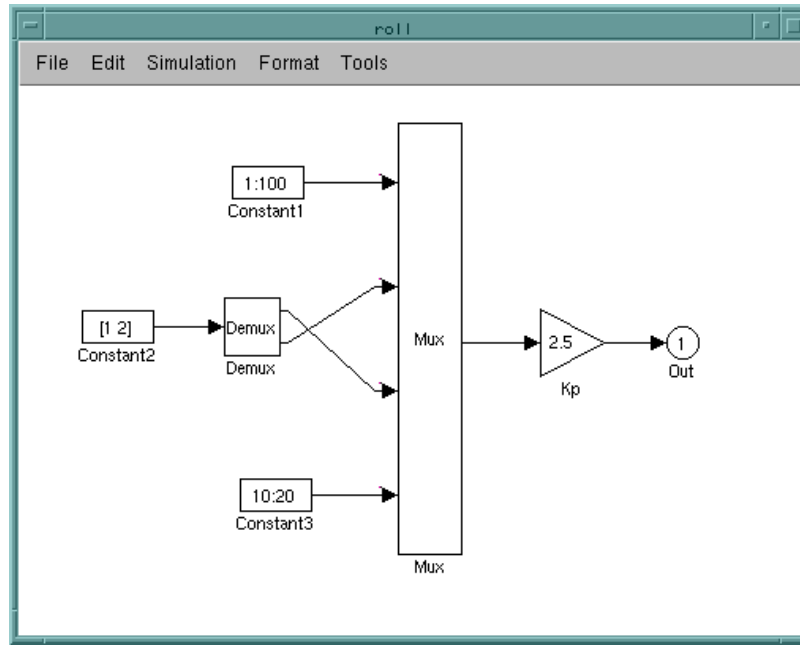


Figure 4-3: Example of Loop Rolling

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 `RollRegions` (specified in the `.rtw` file) and `RollThreshold` (specified at the command line). If there are any regions in `RollRegions` that are greater than the value specified by `RollThreshold`, then those regions will roll. However, any regions in `RollRegions` 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                                                  ┌─┐ ┌─┐ ┌─┐ ┌──┐
      .                                            │ │ │ │ │ │ │  │
      .                                            │ │ │ │ │ │ │  │
      .                                            1 2 3 4
                        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 Target Language Compiler 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 4-1 contains the valid variables assigned to `rollVars`.

Table 4-1: Roll Table Variables

Block	Variable	Description
`Inputs`	`U` `ui`	All inputs input `i`
`Outputs`	`Y` `yi`	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 = RollRegions, lcv = RollThreshold, block, "Roller",
rollVars
```

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

Miscellaneous Topics

Configurable Target Language Compiler Options

See Table 3-2 in Chapter 3 of the *Real-Time Workshop User's Guide* for a list of configurable Target Language Compiler options.

Matrix Parameters in Real-Time Workshop

MATLAB matrices are the transpose of Real-Time Workshop matrices, with the exception of noninlined S-function blocks, which use the MATLAB representation. MATLAB uses column-major ordering and Real-Time Workshop 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 {              /* noninlined S-function */
    real_T matSize[2];
    real_T mat[3][2];
  } sfuncBlock;

struct  {   /* any non S-function block or inlined S-function*/
    real_T mat2[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`, that

```
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`.

5

Target Language Compiler Function Library Reference

Introduction

This chapter provides an alphabetical list of Target Language Compiler functions that are useful for inlining S-functions. The TLC files contain many other library functions, but you should use only the functions that are documented in these reference pages for development. Undocumented functions may change significantly from release to release. A table of obsolete functions and their replacements are shown in "Obsolete Functions".

Reference Page Format

The format of each reference page is as follows:

Function

The function name.

```
<Name of function>
```

Purpose

A short description of what the function does.

Syntax

The syntax for calling the function.

```
<Name of function>(<arg1>, <arg2>, ...)
```

Arguments

A description of arguments to the function.

`<arg1>`: Description of Argument 1

`<arg2>`: Description of Argument 2

.

.

.

Returns

A list of possible values returned by the function.

Description
A detailed description of the function's behavior.

Example
An example of how to use the function.

Note
Any special caveats or warnings about the function.

See Also
Other functions that may be related or useful.

Common Function Arguments
Several functions take similar or identical arguments. To simplify the reference pages, some of these arguments are documented in detail here instead of in the reference pages.

The arguments listed below are used in many functions related to loop rolling. See Chapters 2 and 3 for more information about loop rolling.

- `ucv`: User-control variable string.

  This is an advanced feature that overrides the `lcv` and (the index part of) the `idx` arguments. Generally, this argument is specified as `NULL ""`.
- `lcv`: Loop-control variable string.

  This argument is generally produced by the `%roll` directive and should be passed directly to this function. (The Target Language Compiler sets `lcv` to `""` when a `%roll` is being inlined.) Otherwise, the `lcv` will be set to the loop control variable, such as `"i1"`, indicating that the current pass through the `%roll` is being placed in a `for`-loop. When it is not `NULL`, the `lcv` argument overrides `idx`. See the description of the `%roll` directive in Chapter 2 for more detailed information on the `lcv`.
- `idx`: An overloaded argument specifying the index into a real or complex block input signal.

  This argument is explained in the following examples.

Example 1

Assume the following:

1. Element 3
2. `ucv = lcv = ""`

The following table shows:

- Values of `idx`
- Whether `idx` is complex
- What the function that uses `idx` returns
- An example of a returned variable
- Data type of the returned variable

Note that "container" refers to the data type that encapsulates both the real and imaginary parts of the number.

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

Example 2

Assume the following:

1. Element 3
2. (`ucv = "i"` AND `lcv == ""`) OR (`ucv = ""` AND `lcv = "i"`)

The following table shows values of `idx`, whether `idx` is complex, and what the function that uses `idx` returns.

idx	Complex	Function Returns
`"re3"`	yes	Real part of element i
`"im3"`	yes	Imaginary part of element i
`"3"`	yes	Complex container of element i
`3`	yes	Complex container of element i
`"re3"`	no	Element i
`"im3"`	no	`""`
`"3"`	no	Element i
`3`	no	Element i

Notes

- The vector index is only added for wide signals.
- If `ucv` is not `NULL`, then the `ucv` is used instead of `idx` in the above examples and both `lcv` and `idx` are ignored.
- If `ucv` is `NULL` but `lcv` is not `NULL`, then this function returns `"&y%<portIdx>[%<lcv>]"` and `idx` is ignored.
- It is assumed here that the roller has appropriately declared and initialized the local variable `"y%<portIdx>"`.

Obsolete Functions

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

Obsolete Function	Equivalent Replacement Function
`LibBlockOutportLocation`	`LibBlockDstSignalLocation`
`LibCacheGlobalPrmData`	Use the block function `BlockInstanceData` (see TLC manual for more information on how to use this function)
`LibContinuousState`	`LibBlockContinuousState`
`LibControlPortInputSignal`	`LibBlockSrcSignalLocation`
`LinConvertZCDirection`	Function not used in the Real-Time Workshop
`LibDataInputPortWidth`	`LibBlockInputSignalWidth`
`LibDataOutputPortWidth`	`LibBlockOutputSignalWidth`
`LibDefinePWork`	Specifying `PWork` names is now supported via the `mdlRTW` function in your `C-MEX` S-Function
`LibDefineIWork`	Specifying `IWork` names is now supported via the `mdlRTW` function in your `C-MEX` S-Function
`LibDefineRWork`	Specifying `RWork` names is now supported via the `mdlRTW` function in your `C-MEX` S-Function
`LibDiscreteState`	`LibBlockDiscreteState`
`LibExternalResetSignal`	`LibBlockInputSignal`
`LibMapSignalSource`	`FcnMapDataTypedSignalSource`

Obsolete Function	Equivalent Replacement Function
`LibMaxBlockIOWidth`	Function not used in the Real-Time Workshop
`LibMaxDataInputPortWidth`	Function not used in the Real-Time Workshop
`LibMaxDataOutputPortWidth`	Function not used in the Real-Time Workshop
`LibPathName`	`LibGetBlockPath`, `LibGetFormattedBlockPath`
`LibPrevZCState`	`LibBlockPrevZCState`
`LibRenameParameter`	Specifying parameter names is now supported via the `mdlRTW` function in your `C-MEX` S-Function

Target Language Compiler Functions

The rest of this chapter lists alphabetically and describes all the Target Language Compiler library functions available.

LibAddIdentifier

Purpose

Add an identifier/value pair to a specified scope.

Syntax

```
%<LibAddIdentifier(rec, name, value)>
```

Arguments

`rec`
Record reference.

`name`
String name of the identifier to add to the record reference.

`value`
Value of the identifier.

Description

`LibAddIdentifier` adds an identifier/value pair to a specified scope.

If you add a duplicate identifier with the same `name` but a different `value`, you will get an error message.

Adding a duplicate identifier with the same `name` and `value` has no effect.

Purpose

Add an identifier/value pair to the global `CompiledModel` record.

Syntax

```
%<LibAddToCompiledModel(name, value)>
```

Arguments

`name`
Name of the identifier to add to the `CompiledModel` record.

`value`
Value of the identifier.

Description

`LibAddToCompiledModel` adds an identifier/value pair to the global `CompiledModel` record.

If you add a duplicate identifier with the same `name` but different `value`, you will get an error message.

Adding a duplicate identifier with the same `name` and `value` has no effect.

LibBlockContinuousState

Purpose

Determine a block's continuous states.

Syntax

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

Arguments

`ucv`
User-control variable string.

`lcv`
Loop-control variable string.

`idx`
Integer index into the block's state vector.

Returns

Block's continuous states.

The returned string is composed of four parts: `<vect>`, `<ioq>`, `<id>`, and `<index>`.

The first part, `<vect>`, is determined by the current code format or by loop rolling. Its possible values are:

- `rtX_Cont` for the S-Function code format
- `rtX` for all other code formats
- `xc` inside rolled loops

The second part, `<ioq>`, is an I/O qualifier (or selection operator) that is also determined by the code format. Its possible values are:

- `->` for the S-Function code format
- `->c.` for the `RealTimeMalloc` code format
- `.c.` for all other code formats
- An empty string inside rolled loops

The third part, `<id>`, is the unique identifier assigned to this signal. This identifier is obtained from either the signal label specified in the Simulink diagram or, if a signal label is not present, the name of the block from which this signal originates. The `<id>` is the empty string when you call `LibBlockContinuousState` inside a rolled loop.

The final part, `<index>`, is the index into the appropriate element and/or the real/imaginary part of the input signal, if it is either wide and/or complex. Its exact value depends on `ucv`, `lcv`, and `idx`.

Description

`LibBlockContinuousState` returns the block's discrete states.

See Also

`LibBlockDiscreteState`

LibBlockDiscreteState

Purpose

Determine a block's discrete states.

Syntax

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

Arguments

`ucv`
User-control variable string.

`lcv`
Loop-control variable string.

`idx`
Integer index into the block's state vector.

Returns

Block's discrete states.

The returned string is composed of four parts: `<vect>`, `<ioq>`, `<id>`, and `<index>`.

The first part, `<vect>`, is determined by the current code format or by loop rolling. Its possible values are:

- `rtX_Cont` for the S-Function code format
- `rtX` for all other code formats
- `xc` inside rolled loops

The second part, `<ioq>`, is an I/O qualifier (or selection operator) that is also determined by the code format. Its possible values are:

- `->` for the S-Function code format
- `->c.` for the `RealTimeMalloc` code format
- `.c.` for all other code formats
- An empty string inside rolled loops

The third part, `<id>`, is the unique identifier assigned to this signal. This identifier is obtained from either the signal label specified in the Simulink diagram or, if a signal label is not present, the name of the block from which this signal originates. The `<id>` is the empty string when you call `LibBlockDiscreteState` inside a rolled loop.

The final part, `<index>`, is the index into the appropriate element and/or the real/imaginary part of the input signal, if it is either wide and/or complex. Its exact value depends on `ucv`, `lcv`, and `idx`.

Description

`LibBlockDiscreteState` returns the Block's discrete states.

See Also

`LibBlockContinuousState`

LibBlockInputSignal

Purpose

Determine reference to a block's input.

Syntax

```
%<LibBlockInputSignal(portIdx, ucv, lcv, idx)>
```

Arguments

`portIdx`
Integer input port index, starting from 0, `enable`, or `trigger`.

`ucv`
User-control variable string.

`lcv`
Loop-control variable string.

`idx`
Integer index into signal vector.

Returns

`LibBlockInputSignal` returns four parts to the signal reference: `<vect>`, `<ioq>`, `<id>`, `<index>`.

The first part, <vect>, depends on where this signal is declared, which in turn is determined by the attributes of the driving block and the output port from where this signal originates. The possibilities are:

- `rtU`: External inputs vector (driven by a inport block not in a subsystem)
- `rtX`: States vector (output from the state port of driving block)
- `rtB`: Block I/O vector, declared globally
- `rtb_`: Block I/O vector, declared locally
- `rtC`: Const Block I/O vector (driven by an invariant block)
- `rtC_`: #define'd Const Block I/O (driven by an invariant block) {FALSE, "0", "0.0", "`rt<dType>GROUND`", "`rt<dType>GROUND_Complex`"}: Grounded or unconnected input where `dType` is the input port data type.
- `u`: Inside a rolled loop

The second part, `<ioq>`, is the I/O qualifier (or the selection operator), which is largely determined by the code format. The possibilities are:

- An empty string inside a rolled loop
- `.`: RealTime and Embedded-C code formats
- `->`: `RealTimeMalloc` and S-Function code formats

The third part, <id>, is the unique identifier assigned to this signal. This identifier is obtained from either the signal label specified in the Simulink diagram or, if a signal label is not present, the name of the block from which this signal originates.

For invariant block I/O signals, whose value is set by a `#define` statement, an additional _ is appended between the <id> and <index> positions.

The final part, <index>, is the index into the appropriate element and/or real/imaginary part of the input signal, if it is either wide and/or complex. This depends on `ucv`, `lcv`, and `idx`.

If you combine all of them, you get strings such as `rtB.s7_Gain1[2]`, where:

- `rtB` is the <vect> part
- "." is the I/O qualifier
- `s7_Gain1` is the <id> part (`s7` represents Subsystem 7 while `Gain1` is the name of the block)
- `[2]` is the <index> part.

Description

`LibBlockInputSignal` returns the appropriate reference to a block input signal, based on the:

- input port number (`portIdx`)
- user-control variable (`ucv`)
- loop-control variable (`lcv`)
- signal index (`idx`)
- location of the signal.

Note You should not use this function to access the address of an input signal. The Real-Time Workshop tracks when a variable is accessed by its address. If you access the address of the signal via `LibBlockInputSignal` for invariant block I/O signals whose value is set by a `#define` statement, the result is a reference to a number.

```
%assign u = LibBlockInputSignal(0, "", lcv, sigIdx)
x = &%<u>;
```

If %<u> refers to an invariant signal with a value of 4.95, the statement (after being processed by the preprocessor) would read

```
x = &4.95;
```

If, instead, the input signal sources are connected to ground, the statement could come out as

```
x = &0.0;
```

To avoid these situations, use `LibBlockInputSignalAddr()`:

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

Example

Example 1

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

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

You can use this for a block with a single input port and a single output port.

Example 2

This example is more complicated: a block with multiple input ports where each port can be wide and a single, 1-wide output port. The output signal is the sum of the squares of all the input signals.

```
%assign y = LibBlockOutputSignal(0, "", "", 0)
y = 0;

%assign rollVars = ["U"]
%foreach port = block.NumDataInputPorts - 1
   %roll sigIdx = RollRegions, lcv = RollThreshold, ...
    block,"Roller", rollVars
      %assign u = LibBlockInputSignal(port, "", lcv, sigIdx)
      y += %<u> * %<u>;
   %endroll
%endforeach
```

Since the first parameter of `LibBlockInputSignal` is 0-indexed, you must index the `foreach` loop to start from 0 and end at `NumDataInputPorts-1`.

See Also

`LibBlockInputSignalAddr`, `LibBlockOutputSignal`, `LibBlockOutputSignalAddr`, `LibBlockParameter`, `LibBlockParameterAddr`.

LibBlockInputSignalAddr

Purpose

Determine the memory address of an input signal.

Syntax

```
%<LibBlockInputSignalAddr(portIdx, ucv, lcv, idx)
```

Arguments

`portIdx`
Integer input port index, starting from 0, `enable`, or `trigger`.

`ucv`
User-control variable string.

`lcv`
Loop-control variable string.

`idx`
Integer offset into block signal.

Returns

Address of a block input signal. See `LibBlockInputSignal` for a description of how the name of the returned signal is constructed. Note that an ampersand, "&", is added to the beginning of the name.

Description

`LibBlockInputSignalAddr` returns the address of a block input signal. If you access the address of an invariant signal, it will be declared constant instead of being defined by a `#define` statement.

Note Unlike `LibBlockInputSignal`, the last argument, `idx`, is not overloaded. Hence, if the input signal is complex, the address of the complex container is returned.

Example

To get the address of a wide input signal and pass it to a user-function for processing, you could use:

```
%assign u = LibBlockInputSignalAddr(0, "", "", 0)
%assign y = LibBlockOutputSignal(0, "", "", 0)
y= myfcn (%<u>);
```

See Also

`LibBlockInputSignal`, `LibBlockOutputSignalAddr`, `LibBlockParameterAddr`.

Purpose

Determine if an input signal's buffer is the destination for the output signal of any of the output ports of the block.

Syntax

```
%<LibBlockInputSignalBufferDestPort(portIdx)>
```

Arguments

`portIdx`
Integer input port index, starting from 0.

Returns

Destination port of an input signal. If the input buffer is not being reused, this function returns -1.

Description

If an input port and some output port of a block are not test points and the input port can be overwritten, then the output port may reuse the same buffer as the input port. In this case, `LibBlockInputSignalBufferDstPort` returns the index of the output port that reuses the specified input port's buffer.

If none of the block's output ports reuse the specified input port buffer, then this function returns -1.

This function is the Target Language Compiler implementation of the Simulink macro `ssGetInputPortBufferDstPort`.

Example

You have a block that has two input ports, both of which receive a complex number in a vector of width 2. The block outputs the product of the two complex numbers.

```
%assign u1r = LibBlockInputSignal (0, "", "", 0)
%assign u1i = LibBlockInputSignal (0, "", "", 1)
%assign u2r = LibBlockInputSignal (1, "", "", 0)
%assign u2i = LibBlockInputSignal (1, "", "", 1)
%assign yr  = LibBlockOutputSignal (0, "", "", 0)
%assign yi  = LibBlockOutputSignal (0, "", "", 1)

%if (LibBlockInputSignalBufferDstPort(0) != -1)
   %% The first input is going to get overwritten by yr so
   %% we need to save the real part in a temporary variable.
   {
   real_T tmpRe = %<u1r>;
   %assign u1r = "tmpRe";
%endif
```

```
%<yr> = %<u1r> * %<u2r> - %<u1i> * %<u2i>;
%<yi> = %<u1r> * %<u2i> + %<u1i> * %<u2r>;

%if (LibBlockInputSignalBufferDstPort(0) != -1)
   }
%endif
```

Note that in this case, we could have equivalently used (`LibBlockInputSignalBufferDstPort(0) == 0`) as the boolean condition for the `%if` statements since there is only one output port.

See Also

`LibBlockOutputSignalWidth`

Purpose

Determine the width of an input port.

Syntax

```
%<LibBlockInputSignalWidth(portIdx)>
```

Arguments

`portIdx`
Integer input port index, starting from 0.

Returns

Width of input port.

Description

`LibBlockInputSignalWidth` returns the width of an input port.

See Also

`LibBlockOutputSignalWidth`

LibBlockIWork

Purpose

Determine appropriate `IWork` element.

Syntax

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

Arguments

`iwork`
Reference to `IWork`.

`ucv`
User-control variable string.

`lcv`
Loop-control variable string.

`idx`
Integer index into `IWork` vector.

Returns

The appropriate `IWork` element.

Description

This function returns the appropriate `IWork` element.

Purpose

Determine the matrix parameter of a block.

Syntax

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

Arguments

`param`
Reference to a block parameter.

`rucv`
Row user-control variable string.

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

`ridx`
Row index into the parameter matrix.

`cucv`
Column user-control variable string.

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

`cidx`
Column index into the parameter matrix (overloaded for complex signals)

Returns

Matrix parameter.

Description

`LibBlockMatrixParameter` returns the appropriate matrix parameter for a block given the row and column user-control variables, loop-control variables, and indices.

Note that loop rolling is currently not supported, and will generate an error if requested (i.e., if `rlcv` or `clcv` is not null).

The row and column index arguments are similar to the arguments for `LibBlockParameter`. The column index (`cidx`) is overloaded to handle complex numbers.

See Also

`LibBlockParameter`, `LibBlockMatrixParameterAddr`.

LibBlockMatrixParameterAddr

Purpose

Determine the address of a matrix parameter.

Syntax

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

Arguments

`param`
Reference to a block parameter.

`rucv`
Row user-control variable string.

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

`ridx`
Row index into parameter matrix.

`cucv`
Column user-control variable string.

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

`cidx`
Column index into parameter matrix.

Returns

Address of a matrix parameter.

Description

`LibBlockMatrixParameterAddr` returns the address of a matrix parameter. Note that loop rolling is currently not supported, and generates an error if requested (i.e., `rlcv` or `clcv` should not be an empty string).

Note that `cidx` is not overloaded for complex signals.

See Also

`LibBlockMatrixParameter`, `LibBlockMatrixParameterValue`.

Purpose

Determine the numeric value of a matrix parameter and format it according to its data type.

Syntax

```
%<LibBlockMatrixParameterFormattedValue(param, ridx, cidx)>
```

Arguments

`param`
Reference to a block parameter.

`ridx`
Row index into the parameter matrix.

`cidx`
Column index into the parameter matrix (overloaded for complex signals).

Returns

Value of a matrix parameter as a string, formatted according to its data type.

Description

`LibBlockMatrixParameterFormattedValue` returns the value of a matrix parameter.

See Also

`LibBlockMatrixParameterAddr`, `LibBlockMatrixParameter`

LibBlockMatrixParameterValue

Purpose

Determine the numeric value of a matrix parameter.

Syntax

```
%<LibBlockMatrixParameterAddr(param, ridx, cidx)>
```

Arguments

`param`
Reference to a block parameter.

`ridx`
Row index into the parameter matrix.

`cidx`
Column index into the parameter matrix (overloaded for complex signals).

Returns

Numeric value of a matrix parameter.

Description

`LibBlockMatrixParameterValue` returns the numeric value of a matrix parameter.

See Also

`LibBlockMatrixParameterValue`, `LibBlockMatrixParameterFormattedValue`

Purpose

Determine block mode.

Syntax

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

Arguments

`ucv`
User-control variable string.

`lcv`
Loop-control variable string.

`idx`
Mode index.

Returns

Block mode.

Description

`LibBlockMode` determines the block mode based on `ucv`, `lcv`, and `idx`.

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

Determine the reference to a block's output.

Syntax

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

Arguments

`portIdx`
Integer output port index, starting from 0.

`ucv`
User-control variable string.

`lcv`
Loop-control variable string.

`idx`
Integer index into signal vector.

Returns

Reference to a block's output.

`LibBlockOutputSignal` returns four parts to the signal reference: `<vect>`, `<ioq>`, `<id>`, `<index>`.

The first part, <vect>, depends on where this signal is declared. The possibilities are:

- `rtB`: Block I/O vector, declared globally
- `rtb_`: Block I/O vector, declared locally
- `y`: Inside a rolled loop
- `rtC`: `Const` block I/O vector (outputs an invariant signal)
- `rtC_`: `Const` block I/O whose value is set by a `#define` statement (outputs an invariant signal)

The second part, `<ioq>`, is the I/O qualifier (or the selection operator), which is largely determined by the code format. The possibilities are:

- An empty string inside a rolled loop
- `.`: RealTime and Embedded-C code formats
- `->`: `RealTimeMalloc` and S-Function code formats

The third part, `<id>`, is the uniquified identifier assigned to this signal. This identifier is obtained from either the signal label specified in the Simulink

diagram or, if a signal label is not present, the name of the block from which this signal originates.

For invariant block I/O signals, whose value is set by a `#define` statement, an additional _ is appended between the `<id>` and `<index>` positions.

The final part, `<index>`, is the index into the appropriate element and/or real/imaginary part of the input signal, if it is wide and/or complex. This depends on `ucv`, `lcv`, and `idx`.

Description

Based on the input port number (`portIdx`), the user control variable (ucv), the loop-control variable (`lcv`), the signal index (`idx`), and the location of this output signal, LibBlockOutputSignal returns the appropriate reference to a block output signal.

Note You should not use this function to access the address of an output signal. The Real-Time Workshop tracks when a variable is accessed by its address. If you access the address of the signal via `LibBlockOutputSignal` for invariant block I/O signals whose value is set by a `#define` statement, the result is a reference to a number.

```
%assign y= LibBlockOutputSignal(0, "", lcv, sigIdx)
x = &%<y>;
```

If `%<y>` refers to an invariant signal with a value of 4.95, the statement (after being processed by the preprocessor) would read

```
x = &4.95;
```

Avoid any such situations by using `LibBlockOutputSignalAddr()`:

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

See Also

`LibBlockOutputSignalAddr`, `LibBlockInputSignal`, `LibBlockInputSignalAddr`, `LibBlockParameter`, `LibBlockParameterAddr`.

LibBlockOutputSignalAddr

Purpose

Determine the memory address of an input signal.

Syntax

```
%<LibBlockOutputSignalAddr(portIdx, ucv, lcv, idx)
```

Arguments

`portIdx`
Integer input port index, starting from `0`, `enable`, or `trigger`.

`ucv`
User-control variable string.

`lcv`
Loop-control variable string.

`idx`
Integer offset into block signal.

Returns

Address of a block input signal. See `LibBlockInputSignal` for a description of how the name of the returned signal is constructed. Note that an ampersand, "&", is added to the beginning of the name.

Description

`LibBlockOutputSignalAddr` returns the address of a block output signal. if you access the address of an invariant signal, it will be declared constant instead of being defined by a `#define` statement.

Note Unlike `LibBlockOutputSignal`, the last argument, `idx`, is not overloaded. Hence, if the output signal is complex, the address of the complex container is returned.

Example

To get the address of a wide input signal and pass it to a user-function for processing, you could use:

```
%assign u = LibBlockOutputSignalAddr(0, "", "", 0)
%assign y = LibBlockOutputSignal(0, "", "", 0)
y= myfcn (%<u>);
```

See Also

`LibBlockOutputSignal`, `LibBlockInputSignal`, `LibBlockInputSignalAddr`.

Purpose

Determine if the specified block output is declared in the global I/O vector.

Syntax

```
%<LibBlockOutputSignalIsInBlockIO(portIdx)>
```

Arguments

`portIdx`
Integer output port index, starting from 0.

Returns

1 if the specified block output is in the global block I/O vector; otherwise 0.

Description

`LibBlockOutputSignalIsInBlockIO` returns 1 if the specified block output is declared in the global block I/O vector; otherwise it returns zero.

LibBlockOutputSignalWidth

Purpose

Determine the width of an output port.

Syntax

```
%<LibBlockOutputSignalWidth(portIdx)>
```

Arguments

`portIdx`
Integer output port index, starting from 0.

Returns

Width of output port.

Description

`LibBlockOutputSignalWidth` returns the width of an output port.

See Also

`LibBlockInputSignalWidth`

Purpose

Determine a block parameter.

Syntax

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

Arguments

`param`
Reference to a block parameter name.

`ucv`
User-control variable string.

`lcv`
Loop-control variable string.

`idx`
Integer index into parameter vector.

Returns

Reference to a block parameter.

Description

`LibBlockParameter` returns the appropriate reference to a block parameter, based on:

- Parameter reference (`param`)
- User-control variable (`ucv`)
- Loop-control variable (`lcv`)
- Parameter index (`idx`)
- State of parameter inlining (controlled by the TLC variable `InlineParameters`)

For example, assume you have a gain block (`blk`) with a noncomplex vector parameter, where `Gain[0] = 4.55`:

```
LibBlockParameter(Gain, "", "i", 0)
```

The following table shows the possible results, based on:

- Whether `ucv`, or `lcv`, or both, is a non-empty string (**Rolling**)
- Whether the parameter is inlined (**InlineParameter**)
- Type of signal (**Type**)

Case	Rolling	InlineParameter	Type	Result	Required In Memory
1	0	1	scalar	4.55	no
2	1	1	scalar	4.55	no
3	0	1	vector	4.55	no
4	1	1	vector	p_Gain[i]	yes
5	0	0	scalar	rtP.blk.Gain	yes
6	1	0	scalar	rtP.blk.Gain	yes
7	0	0	vector	rtP.blk.Gain[0]	yes
8	1	0	vector	p_Gain[i]	yes

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

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

See Also

`LibBlockParameterAddr`, `LibBlockInputSignal`, `LibBlockOutputSignal`.

Purpose

Determine the address of a block parameter.

Syntax

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

Arguments

`param`
Reference to a block parameter.

`ucv`
User-control variable string.

`lcv`
Loop-control variable string.

`idx`
Integer index into parameter vector.

Returns

Address of a block parameter.

Description

`LibBlockParameterAddr` returns the address of a block parameter.

If you use the `LibBlockParameterAddr` to access a parameter when the global `InlineParameters` variable is equal to 1, the variable is declared `const` in RAM instead of being inlined.

See Also

`LibBlockParameter`, `LibBlockInputSignalAddr`, `LibBlockOutputSignalAddr`.

LibBlockParameterFormattedValue

Purpose

Determine the value of the parameter and format it according to its data type.

Syntax

```
%<LibBlockParameterFormattedValue(param, idx)>
```

Arguments

`param`
Reference to a block parameter.

`idx`
Integer index into parameter vector.

Returns

Value of the parameter as a string, formatted according to its data type.

Description

`LibBlockParameterAddr` returns the address of a block parameter.

If you use the `LibBlockParameterAddr` to access a parameter when the global `InlineParameters` variable is equal to 1, the variable is declared `const` in RAM instead of being inlined.

See Also

`LibBlockParameterValue`, `LibBlockParameter`.

Purpose

Determine the number of rows and columns in a parameter.

Syntax

```
%<LibBlockParameterSize(param)>
```

Arguments

`param`
Reference to a block parameter.

Returns

Vector of size 2 in the format `[nRows, nCols]`, where `nRows` is the number of rows, and `nCols` is the number of columns.

Description

`LibBlockParameterSize` returns the size of a block parameter.

LibBlockParameterValue

Purpose

Determine the numeric value of a block parameter.

Syntax

```
%<LibBlockParameterValue(param, idx)>
```

Arguments

`param`
Reference to a block parameter.

`idx`
Integer index into parameter vector.

Returns

Numeric value of the parameter.

Description

`LibBlockParameterValue` returns the numeric value of the parameter. You can manipulate this value within the Target Language Compiler.

Note If you access a parameter value using only `LibBlockParameterValue`, it is better to declare this parameter as a `ParamSetting`, since its value is read only during code generation and not during run-time.

Example

If you want to generate code for a different integrator depending on a block's parameter, you could use the following:

```
%assign mode = LibBlockParameterValue(Integrator, 0)
%switch (mode)
   %case 1
       %<CodeForIntegrator1>
       %break
   %case 2
       %<CodeForIntegrator2>
       %break
   %default
       %exit Unrecognized integrator value.
       %break
%endswitch
```

See Also

`LibBlockParameter`

Purpose

Determine appropriate `PWork` element.

Syntax

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

Arguments

`pwork`
Reference to `PWork`.

`ucv`
User-control variable string.

`lcv`
Loop-control variable string.

`idx`
Integer index into `PWork` vector.

Returns

The appropriate `PWork` element.

Description

`LibBlockPWork` returns appropriate `PWork` element.

LibBlockRWork

Purpose

Determine appropriate `RWork` element.

Syntax

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

Arguments

`rwork`
Reference to `RWork`.

`ucv`
User-control variable string.

`lcv`
Loop-control variable string.

`idx`
Integer index into `RWork` vector.

Returns

The appropriate `RWork` element.

Description

`LibBlockRWork` returns appropriate `RWork` element.

Purpose

Determine a block's sample time.

Syntax

```
%<LibBlockSampleTime(block)>
```

Arguments

`block`
Reference to the block record.

Returns

Discrete blocks: A real number greater than zero; the block's actual sample time:

- Continuous blocks: `0.0`
- Triggered blocks: `-1.0`
- Constant blocks: `-2.0`

Description

`LibBlockSampleTime` returns a block's sample time.

LibBlockSrcSignalBlock

Purpose

Determine the source of a block signal.

Syntax

```
%<LibBlockSrcSignalBlock(portIdx, idx)>
```

Arguments

`portIdx`
Integer input port index, starting from 0.

`idx`
Integer index into signal vector.

Returns

Depends on block signal, as shown in the table below.

Block Signal	Returns
Unique block output or state	`[systemIdx, blockIdx]`
External input (root inport)	`ExternalInput`
Unconnected or connected to ground	`Ground`
Function-call output	`FcnCall`
Not unique (i.e., if the signal is a source to a `Merge` block or if the signal is reused due to block I/O optimization)	`0`

Description

`LibBlockSrcSignalIsDiscrete` returns the source of a block signal. You can use the returned values in the Target Language Compiler, but they are not suitable for use in C code.

Purpose

Determine if the source of a block signal is discrete.

Syntax

```
%<LibBlockSrcSignalIsDiscrete(portIdx, idx)>
```

Arguments

`portIdx`
Integer input port index, starting from 0.

`idx`
Integer index into signal vector.

Returns

1 if source signal is discrete; otherwise 0 (or if the signal cannot be uniquely determined). The signal cannot be uniquely determined if it is a merged or reused signal (i.e., the source is a `Merge` block or if the signal has been reused due to block I/O optimization).

Description

`LibBlockSrcSignalIsDiscrete` determines if the source of a block signal is discrete.

See Also

`LibBlockSrcSignalBlock`

LibBlockSrcSignalIsGlobalAndModifiable

Purpose

Determine if the source of a block signal is in global memory.

Syntax

```
%<LibBlockSrcSignalIsGlobalAndModifiable(portIdx, idx)>
```

Arguments

`portIdx`
Integer input port index, starting from 0.

`idx`
Integer index into signal vector.

Returns

1 if the block's specified source signal is in global memory.

Description

`LibBlockSrcSignalIsIGlobalAndModifiable` determines if the block's specified source signal is in global memory. If the signal is in global memory, it satisfies the following conditions:

- It is visible everywhere in the generated code.
- It can be referenced by its address.
- Its value can change (i.e., it is not declared as a constant).

Purpose

Determine if the source of a block signal is discrete.

Syntax

```
%<LibBlockSrcSignalIsInvariant(portIdx, idx)>
```

Arguments

`portIdx`
Integer input port index, starting from 0.

`idx`
Integer index into signal vector.

Returns

1 if source signal is invariant; otherwise 0.

Description

`LibBlockSrcSignalIsInvariant` determines if the specified block input signal is invariant (i.e., the block I/O signal does not change). For example, a source block with a constant TID (or equivalently, an infinite sample time) outputs an invariant signal.

LibCacheDefine

Purpose

Cache #define statements to include in `<model>.h`.

Syntax

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

Arguments

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

Description

You can call `LibCacheDefine` 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 in `<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)>
```

Purpose

Cache statements that you want to declare `extern`.

Syntax

```
%<LibCacheExtern(buffer)>
```

Arguments

`buffer`
Buffer of `extern` statements to be cached.

Description

You can call `LibCacheExtern` from inside `BlockTypeSetup` to cache an `extern` statement. Each call to this function appends your buffer to the existing cache buffer. The `extern` statements are placed in `<model>.h` or in `xxx` if the source file splits.

Example

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

LibCacheFunctionPrototype

Purpose

Cache a function prototype to include in `<model>.h`.

Syntax

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

Arguments

`buffer`
Buffer of function prototypes to be cached.

Description

You can call `LibCacheFunctionPrototype` 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 in `<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

Cache `#include` statements to include in `<model>.h`.

Syntax

```
%<LibCacheIncludes(buffer)>
```

Arguments

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

Description

You can call `LibCacheInclude` 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 in `<model>.h`.

Example

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

LibCacheNonFiniteAssignment

Purpose

Cache a nonfinite assignment that needs to be placed in the registration function.

Syntax

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

Arguments

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

Description

You can call `LibCacheNonFiniteAssignment` 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
%<LibBlockParameter(Gain, "","",0)> =
       %<LibRealNonFinite(inf)>;
%<LibBlockParameter(Gain, "","",1)> =
       %<LibRealNonFinite(nan)>;
%<LibBlockParameter(Gain, "","",2)> =
       %<LibRealNonFinite(-inf)>;
%closefile buffer
%<LibCacheNonFiniteAssignment(buffer)>
```

Purpose

Cache `typedef` statements to include in `<model>.h`.

Syntax

```
%<LibCacheTypeDefs(buffer)>
```

Arguments

`buffer`
Buffer of `typedef` statements to be cached.

Description

You can call `LibCacheInclude` from inside `BlockTypeSetup` to cache `typedef` declarations. Each call to this function appends your buffer to the existing cache buffer. The `typedef` statements are placed in `<model>.h`.

LibCallFCSS

Purpose

Generate a function-call for an inlined S-Function.

Syntax

```
%<LibCallFCSS(system, simObject, fcnPortE1, tidVal)>
```

Arguments

`system`
System record of the function-call subsystem that you are calling.

`simObject`
Variable name for the `SimStruct`, usually `tSimStruct`.

`fcnPortE1`
Function-call input port element to which the S-Function is connected.

`tidVal`
S-Function TID (`%<LibTID()>`).

Returns

Call to function-call subsystem with the appropriate number of arguments or inlined code. The returned string is determined by the current code format.

Description

`LibCallFCSS` is used by inlined S-Functions to make a function call. This function returns the call to the function-call subsystem with the appropriate number of arguments or inlined code. An S-function can execute a function-call subsystem only via its first output port.

See the `SFcnSystemOutputCall` record in the `model.rtw` file.

Example

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

`BlockToCall` and `FcnPortElement` are elements of the `SFcnSystemOutputCall` record.

`System` is a record within the global `CompiledModel` record.

This example is from the following file:

```
matlabroot/toolbox/simulink/blocks/tlc_c/fncallgen.tlc
```

LibDataStoreMemory

Purpose

Determine 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.

Purpose

Determine the full path name for a block.

Syntax

```
%<LibGetBlockPath(block)>
```

Arguments

`block`
Reference to a block record.

Returns

The full pathname string of a block, including carriage or hard returns and other special string sequences that may be in the name.

Description

`LibGetBlockPath` returns the full pathname string for a block record, including carriage or hard returns and other special string sequences that may be in the name. Currently, the only other special string sequences defined are `/*` and `*/`.

The full block pathname string is useful when you access blocks from MATLAB. For example, you can use the full block name with `open_system()` to match the Simulink pathname exactly.

See Also

`LibGetFormattedBlockPath`

LibGetFormattedBlockPath

Purpose

Determine the full pathname for a block and remove any special characters in it.

Syntax

```
%<LibGetFormattedBlockPath(block)>
```

Arguments

`block`
Reference to a block.

Returns

The full pathname string of a block, without any special string sequences.

Description

`LibGetFormattedBlockPath` returns the fullpath name string of a block, without any special characters.

Special Character	Converted To
Carriage return (hard return)	Space
/*	/+
*/	+/

You can use the returned block name string in comments or in generated code, on a single line.

See Also

`LibGetBlockPath`

Purpose

Determine the model task identifier (TID) for a local S-Function task identifier.

Syntax

```
%<LibGetGlobalTIDFromLocalSFcnTID(SFcnTID)>
```

Arguments

`SFcnTID`

- For block-based sample times, an integer corresponding to local S-Function sample time. If you call `ssSetNumSampleTimes(S,N)`, with `N>1` in `mdlInitialSizes` in an S-Function, then the S-Function has a block-based sample time.
- For port-based sample times, a string giving the input (or output) port index. The string must be in the form `InputPortIdxI` or `OutputPortIdxI`, where `I` is a number ranging from 0 to the number of ports (e.g., `InputPortIdx0` or `OutputPortIdx7`). If you call `ssSetNumSampleTimes(S,PORT_BASED_SAMPLE_TIMES)` in `mdlInitialSizes` in an S-Function, then the S-Function has a port-based sample time.

Returns

Model task identifier (sample time index) for a local S-Function task identifier or port sample time.

Description

`LibGetGlobalTIDFromLocalSFcnTID` returns the model task identifier for a local S-Function task identifier. This function lets you use one function to determine a global TID, independent of port- or block-based sample times.

If you call this function with an integer argument, it is equivalent to the statement

```
SampleTimesToSet[SfcnTID][1]
```

where `SampleTimesToSet` is a matrix that maps local S-Function TIDs to global TIDs.

See Also

`LibGetNumSFcnSampleTimes`, `LibIsSFcnSingleRate`, `LibIsSFcnSampleHit`, `LibIsSFcnSpecialSampleHit`, `LibGetSFcnTIDType`.

Example

Example 1: Multirate Block

```
%assign globalTID = LibGetGlobalTIDFromLocalSFcnTID(2)
or
%assign globalTID = ...
    LibGetGlobalTIDFromLocalSFcnTID("InputPortIdx4")

%assign period  = ...
    CompiledModel.SampleTime[globalTID].PeriodAndOffset[0]
%assign offset  = ...
    CompiledModel.SampleTime[globalTID].PeriodAndOffset[1]
```

Example 2: Inherited Sample-Time Block

```
%switch (LibGetSFcnTIDType(0))
   %case "discrete"
   %case "continuous"
       %assign globalTID = LibGetGlobalTIDFromLocalSFcnTID(2)
       %assign period  = ...
        CompiledModel.SampleTime[globalTID].PeriodAndOffset[0]
       %assign offset  = ...
        CompiledModel.SampleTime[globalTID].PeriodAndOffset[1]
       %break
   %case "triggered"
       %assign period = -1
       %assign offset = -1
       %break
   %case "constant"
       %assign period = rtInf
       %assign offset = 0
       %break
   %default
       %exit Unknown tid type
       %break
%endswitch
```

Purpose

Gets the number of S-Function sample times for a block.

Syntax

```
%<LibGetNumSFcnSampleTimes(block)>
```

Arguments

`block`
Reference to the block record.

Returns

Number of S-Function sample times.

Description

`LibGetNumSFcnSampleTimes` returns the number of S-Function sample times for the referenced block.

See Also

`LibIsSFcnSingleRate`, `LibGetGlobalTIDFromLocalSFcnTID`, `LibIsSFcnSampleHit`, `LibIsSFcnSpecialSampleHit`, `LibGetSFcnTIDType`.

LibGetSFcnTIDType

Purpose

Get the task identifier (TID) type of an S-Function.

Syntax

```
%<LibGetSFcnTIDType(SFcntid)>
```

Arguments

`SFcntid`

- For block-based sample times, an integer corresponding to local S-Function sample time. If you call `ssSetNumSampleTimes(S,N)`, with `N>1` in `mdlInitialSizes` in an S-Function, then the S-Function has a block-based sample time.
- For port-based sample times, a string giving the input (or output) port index. The string must be in the form `InputPortIdxI` or `OutputPortIdxI`, where `I` is a number ranging from 0 to the number of ports (e.g., `InputPortIdx0` or `OutputPortIdx7`). The S-Function has a port-based sample time if you call `ssSetNumSampleTimes(S,PORT_BASED_SAMPLE_TIMES)` in `mdlInitialSizes` in an S-Function.

Returns

- `continuous` if the specified S-Function TID is continuous.
- `discrete` if the specified S-Function TID is discrete.
- `triggered` if the specified S-Function TID is triggered.
- `constant` if the specified S-Function TID is constant.

Description

`LibGetSFcnTIDType` returns a string depending on the S-Function's task identifier. This function is useful in the context of S-Functions with an inherited sample time.

See Also

`LibGetNumSFcnSampleTimes`, `LibIsSFcnSingleRate`, `LibGetGlobalTIDFromLocalSFcnTID`, `LibIsSFcnSampleHit`, `LibIsSFcnSpecialSampleHit`.

Purpose

Returns the Simulink macro to access absolute time and sets the global flag `CompiledModel.NeedAbsoluteTime`.

Syntax

```
%<LibGetT()>
```

Arguments

None.

Returns

Simulink macro to access absolute time, `ssGetT`.

Description

`LibGetT` returns `ssGetT(<name of SimStruct>)` and sets the global `NeedAbsoluteTime` flag.

LibGetTaskTimeFromTID

Purpose

Determine the absolute task time for the block.

Syntax

```
%<LibGetTaskTimeFromTID(block)>
```

Arguments

`block`
Reference to the block record.

Returns

`ssGetTaskTime(S, TID)` if the block is constant; otherwise `ssGetT(S)`. In both cases, `S` is the name of the `SimStruct`.

Description

`LibGetTaskTimeFromTID` returns `ssGetTaskTime(S, TID)` if the block is constant or `ssGetT(S)` if the block is not constant.

Purpose

Place code at the top or bottom of the model's header file (`<model>.h`) by specifying `header` or `trailer`, respectively.

Syntax

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

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
Location in which buffer is placed. Use one of the following:

- `header` to place code at top of file
- `trailer` to place code at bottom of file.

Description

`LibHeaderFileCustomCode` places custom code in the model's header file (`<model>.h`). Each call to this function appends your buffer to the internal cache buffer.

LibIsBlockOutputInBlockIO

Purpose

Determine if the specified block output is declared on the global block I/O vector.

Syntax

```
%<LibIsBlockOutputInBlockIO(block, portNum)>
```

Arguments

`block`
Reference to the block record.

`portNum`
Integer output port number.

Returns

1 if the specified block output is declared on the global block I/O vector; otherwise returns 0.

Description

The `LibIsBlockOutputInBlockIO` returns 1 if the specified block output is declared on the global block I/O vector, and otherwise, it returns 0.

Purpose

Determine if a task identifier (TID) is continuous.

Syntax

```
%<LibIsContinuous(tid)>
```

Arguments

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

Returns

1 if TID is continuous; otherwise 0.

Description

`LibIsContinuous` returns 1 if a TID is continuous; otherwise, it returns 0.

Note that TIDs equal to `triggered` or `constant` are not continuous.

LibIsDiscrete

Purpose

Determine if a task identifier (TID) is discrete.

Syntax

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

Arguments

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

Returns

1 if TID is discrete; otherwise 0.

Description

`LibIsDiscrete` returns 1 if a TID is discrete; otherwise, it returns 0.

Note that TIDs equal to `triggered` or `constant` are not discrete.

Purpose

Determine if an input is empty.

Syntax

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

Arguments

`input`
Input expression to test.

Returns

1 if

- Input is an empty string: `""`
- An empty vector: `[]`, `[[]]`, `[]]`, etc.
- An empty matrix array: `[[] []]`
- 0 otherwise

Description

`LibIsEmpty` determines if an input is empty or null.

Purpose

Compare two expressions (not necessarily of the same type) for equality.

Syntax

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

Arguments

`expr1`
First expression.

`expr2`
Second expression.

Returns

1 if `expr1` equals `expr2`; otherwise 0.

Description

`LibIsEqual` returns 1 if `expr1` equals `expr2`; otherwise, it returns 0.

`LibIsEqual` is different than the simple equality `"=="` because it checks the types of the expressions. If you compare expressions of different types, this function will return 0. That is, `"0"` does not equal `0`. For example, `"0"` does not equal `0`. However, expressions of type `"Number"` can be compared to those of type `"Real"`, and expressions of type `"String"` can be compared to those of type `"Identifier"`.

Purpose

Determine if the number is finite.

Syntax

```
%<LibIsFinite(value)>
```

Arguments

`value`
Any number, including `Inf`, `MinusInf`, and `NaN`.

Returns

1 if the number is finite; otherwise 0.

Description

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

LibIsFirstInitCond

Purpose

Return generated code intended for use in the initialization function and determine, during run-time, whether the initialization function is being called for the first time.

Syntax

```
%<LibIsFirstInitCond(s)>
```

Arguments

`s`
`SimStructVariable` name (usually `%<tSimStruct>`).

Returns

Call to Simulink macro `ssIsFirstInitCond`.

Description

`LibIsFirstInitCond` returns generated code intended for placement in the initialization function. During run-time, this code determines whether the initialization function is being called for the first time.

This function also sets a flag that tells the Real-Time Workshop if it needs to declare and maintain the first-initialize-condition flag. Currently, the Embedded-C code format uses this flag to generate more efficient code.

This function is the implementation of the Simulink macro `ssIsFirstInitCond()`.

Purpose

Determine if the signal is in global memory, i.e., whether the signal can be referenced by address.

Syntax

```
%<LibIsInputSignalGlobalandModifiable(block, port, el)>
```

Arguments

`block`
Reference to the block record.

`port`
Input port number.

`el`
Input port element (should be between `0` and `inputPortWidth - 1`.

Returns

1 if the signal is in global memory; otherwise, it returns 0.

Description

`LibIsInputSignalGlobalandModifiable` returns 1 if the signal is in global memory, and otherwise, it returns 0.

LibIsSampleHit

Purpose

Return code that determines if a sample hit occurred.

Syntax

```
%<LibIsSampleHit(tid)>
```

Arguments

`sti`
Sample time index of block (only relevent for `Zero-Order Hold` and `Unit Delay` blocks). The sample time index is the index of the slower sample time.

`tid`
Task identifier (TID) of block.

The TID is the index of the task with the faster sample time.

Returns

One of the following sample hit macros:

Macro	Condition
`ssIsSampleHit(S, %<tid>, tid)`	Discrete TID
`ssIsContinuousTask(S, tid)`	Continuous TID

Description

`LibIsSampleHit` returns the appropriate TID scope, depending on whether the TID is discrete or continuous.

Note that this function cannot be called for a TID that is neither continuous or discrete.

S-Function blocks should not use this function directly; instead, they should use `LibIsSFcnSpecialHit`.

Purpose

Determine if a sample hit occurs for a local S-Function task identifier (TID).

Syntax

```
%<LibIsSFcnSampleHit(SFcntid)>
```

Arguments

`SFcntid`

- For block-based sample times, an integer corresponding to local S-Function sample time. If you call `ssSetNumSampleTimes(S,N)`, with `N>1` in `mdlInitialSizes` in an S-Function, then the S-Function has a block-based sample time.
- For port-based sample times, a string giving the input (or output) port index. The string must be in the form `InputPortIdxI` or `OutputPortIdxI`, where `I` is a number ranging from 0 to the number of ports (e.g., `InputPortIdx0` or `OutputPortIdx7`). The S-Function has a port-based sample time if you call `ssSetNumSampleTimes(S,PORT_BASED_SAMPLE_TIMES)` in `mdlInitialSizes` in an S-Function.

Returns

1 if a sample hit occurs; otherwise 0.

Description

`LibIsSFcnSampleHit` determines if a sample hit occurs for a local S-Function TID.

Example

Example 1

If you have a multirate S-Function block with four block sample times, then the call `LibIsSFcnSampleHit(2)` will return the code to check for a sample hit on the 3rd S-function block sample time.

Example 2

If you have a multirate S-Function block with three input and three output sample times, the call `LibIsSFcnSampleHit("InputPortIdx0")` returns the code to check for a sample hit on the first input port. The call `LibIsSFcnSampleHit("OutputPortIdx7")` returns the code to check for a sample hit on the eight output port.

See Also

`LibGetNumSFcnSampleTimes`, `LibIsSfcnSingleRate`, `LibGetGlobalTIDFromLocalSFcnTID`, `LibIsSFcnSpecialSampleHit`, `LibGetSFcnTIDType`, `LibIsSampleHit`, `LibIsSpecialSampleHit`.

LibIsSFcnSingleRate

Purpose

Determine if an S-Function is single-rate (has one sample time) or multirate (has multiple sample times).

Syntax

```
%<LibIsSFcnSingleRate(block)>
```

Arguments

`block`
Reference to the block record.

Returns

1 if S-Function is single-rate; otherwise 0.

Description

`LibIsSFcnSingleRate` returns 1 if the S-Function is a single-rate block; otherwise, it returns 0.

See Also

`LibIsRateTransitionBlock`, `LibGetNumSFcnSampleTimes`, `LibGetGlobalTIDFromLocalSFcnTID`, `LibIsSFcnSampleHit`, `LibIsSFcnSpecialSampleHit`.

Purpose

Promote a slow task to a faster task, but run it at the slow rate.

Syntax

```
%<LibIsSFcnSpecialSampleHit(sFcnSTI, SFcnTID)>
```

Arguments

For Multirate S-Function Blocks

`sFcnSTI`
Local S-Function sample time index (sti) of the slow task that is promoted.

`SFcnTID`
Local S-Function task identifer (TID) of the fast task to where the slow task is promoted.

For Single-Rate S-Function Blocks

For single rate S-function blocks using `SS_OPTION_RATE_TRANSITION`, `sfcnSTI` and `sfcnTID` are ignored and you should specify them as "".

The format of `sfcnSTI` and `sfcnTID` must follow that of the argument to `LibIsSFcnSampleHit`.

Returns

Simulink macro to promote a slow task (`sfcnSTI`) into a faster task (`sfcnTID`).

Description

`LibIsSFcnSpecialSampleHit` returns the code that promotes a slow task into a fast task, but still runs at the slow rate. See `simstruc.h`, `ssIsSpecialSampleHit`, and `rt_sim.c`.

This advanced function is specifically intended for use in rate transition blocks.

This function determines the global TID from the S-Function TID and calls `LibIsSpecialSampleHit` using the global TIDs for both the sample time index (sti) and the task identifier (TID). See Chapter 6 of the *Real-Time Workshop User's Guide* for more infomation.

Example

Example 1

A rate transition S-function (one sample time with `SS_OPTION_RATE_TRANSITION`):

```
if (%<LibIsSFcnSpecialSampleHit("","")>) {
```

Example 2

2) A multirate S-function with port-based sample times where the output rate is slower than the input rate (e.g. a zero-order hold operation).

```
if
(%<LibIsSFcnSpecialSampleHit("OutputPortIdx0","InputPortIdx0")>)
 {
```

See Also

`LibGetNumSFcnSampleTimes`, `LibIsSFcnSingleRate`, `LibGetGlobalTIDFromLocalSFcnTID`, `LibIsSFcnSampleHit`, `LibGetSFcnTIDType`, `LibIsSampleHit`, `LibIsSpecialSampleHit`.

Purpose

Determine if the system is single-rate.

Syntax

```
%<LibIsSingleRateSystem(system)>
```

Arguments

`system`
Reference to a Simulink system.

Returns

1 if all specified tasks are equal; otherwise 0.

Description

`LibIsingleRateSystem` returns 1 if the system is a single-rate system; otherwise it returns 0.

A system is considered single-rate if either of the following conditions is satisfied:

- The number of systems tasks is one
- The number of system tasks is two, the system task vector is [0, 1], `CompiledModel.FixedStepOpts.TID01EQ` is true, and the system has no continuous states. (`TID01EQ` is true if the fixed-step size for the continuous task is the same as the sample time for the first discrete task.)

LibIsSpecialSampleHit

Purpose

Return code that determines if a special sample hit occurred.

Syntax

```
%<LibIsSpecialSampleHit(sti, tid)>
```

Arguments

`sti`
Sample time index of block (only relevent for `Zero-Order Hold` and `Unit Delay` blocks). The sample time index is the index of the slower sample time.

`tid`
Task identifier (TID) of block. The TID is the index of the task with the faster sample time.

Returns

Call to Simulink's special sample hit macro, `ssIsSpecialSampleHit`.

Description

`LibIsSpecialSampleHit` returns the appropriate special sample hit macro.

You should use this function with multi-rate models. See Chapter 6 of the Real-Time Workshop User's Guide for more information about multi-rate and multi-tasking models.

S-Function blocks should use `LibIsSFcnSpecialSampleHit` instead of this function.

See Also

`LibIsSFcnSpecialSampleHit`

Purpose

Place declaration statements and executable code inside the registration function.

Syntax

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

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
Location in which buffer is placed. Use one of the following:

- `header` to place buffer at top of function.
- `declaration` to place buffer at top of function (same as specifying `header`).
- `execution` to place buffer at top of function, but after `header`.
- `trailer` to place buffer at bottom of function.

Description

`LibMdlRegCustomCode` places declaration statements and executable code inside the registration function.

This code is output to functions depending on the current code format, as shown in the table below.

Function Name	Code Format
`<model>_initialize`	`Embedded-C`
`<model>_malloc`	`S-Function`
`<model>`	`RealTime, RealTimeMalloc`

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

LibMdlStartCustomCode

Purpose

Place declaration statements and executable code inside the start function.

Syntax

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

Arguments

`buffer`
String buffer to append to internal cache buffer.

`location`
Location in which buffer is placed. Use one of the following:

- `header` to place buffer at top of function.
- `declaration` to place buffer at top of function (same as specifying `header`).
- `execution` to place buffer at top of function, but after `header`.
- `trailer` to place buffer at bottom of function.

Description

`LibMdlStartCustomCode` places declaration statements and executable code inside the start function.

This code is output to functions depending on the current code format, as shown in the table below.

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

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

Purpose

Place declaration statements and executable code inside the terminate function.

Syntax

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

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
Location in which buffer is placed. Use one of the following:

- `header` to place buffer at top of function.
- `declaration` to place buffer at top of function (same as specifying `header`).
- `execution` to place buffer at top of function, but after `header`.
- `trailer` to place buffer at bottom of function.

Description

`LibMdlTerminateCustomCode` places declaration statements and executable code inside the start function.

This code is output to functions depending on the current code format, as shown in the table below.

Function Name	Code Format
`<model>_terminate`	`Embedded-C`
`mdlTerminate`	`S-Function`
`MdlTerminate`	`RealTime, RealTimeMalloc`

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

LibPrmFileCustomCode

Purpose

Place code at the top or bottom of the model's parameter file (`<model>.prm`) by specifying `header` or `trailer`, respectively.

Syntax

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

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
Location in which buffer is placed. Use one of the following:

- `header` to place code at top of file
- `trailer` to place code at bottom of file.

Description

`LibHeaderFileCustomCode` places custom code in the model's parameter file (`<model>.prm`). Each call to this function appends your buffer to the internal cache buffer.

Purpose

Returns the appropriate name for a non-finite variable

Syntax

```
%<LibRealNonFinite(SpecialValue)>
```

Arguments

`SpecialValue`

Can be one of `inf`, `nan`, `-inf`.

Description

Returns appropriate non-finite name and sets the `CompiledModel.NeedRealNonFinite` flag. This routine should always be used instead of hard-coding `%<tInf>`, `%<tMinusInf>`, or `%<tNaN>`.

Example

```
%openfile buffer
%<LibBlockParameter(Gain, "","",0)> =
       %<LibRealNonFinite(inf)>;
%<LibBlockParameter(Gain, "","",1)> =
       %<LibRealNonFinite(nan)>;
%<LibBlockParameter(Gain, "","",2)> =
       %<LibRealNonFinite(-inf)>;
%closefile buffer
%<LibCacheNonFiniteAssignment(buffer)>
```

LibRegFileCustomCode

Purpose

Place code at the top or bottom of the model's registration file (`<model>.reg`) by specifying `header` or `trailer`, respectively.

Syntax

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

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
Location in which buffer is placed. Use one of the following:

- `header` to place code at top of file
- `trailer` to place code at bottom of file.

Description

`LibSourceFileCustomCode` places custom code in the model's registration file (`<model>.reg`). Each call to this function appends your buffer to the internal cache buffer.

Purpose

Place code at the top or bottom of the model's source file (`<model>.c`) by specifying `header` or `trailer`, respectively.

Syntax

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

Arguments

`buffer`
Buffer to append to internal cache buffer.

`location`
Location in which buffer is placed. Use one of the following:

- `header` to place code at top of file
- `trailer` to place code at bottom of file.

Description

`LibSourceFileCustomCode` places custom code in the model's source file (`<model>.c`). Each call to this function appends your buffer to the internal cache buffer.

Note if you expect the file to split, be careful when you place code in `<model>.c`. When code is needed in each split file, place it in `<model>.h` instead of `<model>.c`.

LibSystemDerivativeCustomCode

Purpose

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

Syntax

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

Arguments

`system`
Reference to a system.

`buffer`
String buffer to append to internal cache buffer.

`location`
Location in which buffer is placed. Use one of the following:

- `header` to place buffer at top of function.
- `declaration` to place buffer at top of function (same as specifying `header`).
- `execution` to place buffer at top of function, but after `header`.
- `trailer` to place buffer at bottom of function.

Description

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

This code is output to functions for the root system depending on the current code format, as shown in the table below.

Function Name	Code Format
`mdlDerivatives`	`S-Function`
`MdlDerivatives`	`RealTime, RealTimeMalloc`

This function is not relevant for the Embedded C-code format since blocks with continuous states cannot be used. If you try to add code to a subsystem that does not have any continuous states, you will get an error.

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

Purpose

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

Syntax

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

Arguments

`system`
Reference to a system.

`buffer`
String buffer to append to internal cache buffer.

`location`
Location in which buffer is placed. Use one of the following:

- `header` to place buffer at top of function.
- `declaration` to place buffer at top of function (same as specifying `header`).
- `execution` to place buffer at top of function, but after `header`.
- `trailer` to place buffer at bottom of function.

Description

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

If you attempt to add code to a subsystem that does not have a disable function, this function generates an error .

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

LibSystemEnableCustomCode

Purpose

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

Syntax

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

Arguments

`system`
Reference to a system.

`buffer`
String buffer to append to internal cache buffer.

`location`
Location in which buffer is placed.Use one of the following:

- `header` to place buffer at top of function.
- `declaration` to place buffer at top of function (same as specifying `header`).
- `execution` to place buffer at top of function, but after `header`.
- `trailer` to place buffer at bottom of function.

Description

`LibSystemEnableCustomCode` places declaration statements and executable code inside the system's enable function.

If you attempt to add code to a subsystem that does not have an enable function, this function generates an error.

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

Purpose

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

Syntax

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

Arguments

`system`
Reference to a system.

`buffer`
String buffer to append to internal cache buffer.

`location`
Location in which buffer is placed. Use one of the following:

- `header` to place buffer at top of function.
- `declaration` to place buffer at top of function (same as specifying `header`).
- `execution` to place buffer at top of function, but after `header`.
- `trailer` to place buffer at bottom of function.

Description

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

This code is output to functions for the root system depending on the current code format, as shown in the table below.

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

The code for a subsystem is output to the subsystem's initialization function.

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

LibSystemOutputCustomCode

Purpose

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

Syntax

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

Arguments

`system`
Reference to a system.

`buffer`
String buffer to append to internal cache buffer.

`location`
Location in which buffer is placed. Use one of the following:

- `header` to place buffer at top of function.
- `declaration` to place buffer at top of function (same as specifying `header`).
- `execution` to place buffer at top of function, but after `header`.
- `trailer` to place buffer at bottom of function.

Description

`LibSystemOutputCustomCode` places declaration statements and executable code inside the system's output function.

This code is output to functions depending on the current code format, as shown in the table below.

Function Name	Code Format
`<model>_step`	`Embedded-C` (`CombineOutputUpdateFcns` is 1)
`<model>_output`	`Embedded-C` (`CombineOutputUpdateFcns` is 0)
`mdlOutputs`	`S-Function`
`MdlOutputs`	`RealTime, RealTimeMalloc`

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

Purpose

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

Syntax

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

Arguments

`system`
Reference to a system.

`buffer`
String buffer to append to internal cache buffer.

`location`
Location in which buffer is placed. Use one of the following:

- `header` to place buffer at top of function.
- `declaration` to place buffer at top of function (same as specifying `header`).
- `execution` to place buffer at top of function, but after `header`.
- `trailer` to place buffer at bottom of function.

Description

`LibSystemUpdateCustomCode` places declaration statements and executable code inside the system's update function.

This code is output to functions depending on the current code format, as shown in the table below.

Function Name	Code Format
`<model>_step`	`Embedded-C` (`CombineOutputUpdateFcns` is 1)
`<model>_output`	`Embedded-C` (`CombineOutputUpdateFcns` is 0)
`mdlUpdate`	`S-Function`
`MdlUpdate`	`RealTime, RealTimeMalloc`

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

LibSystemUserCodeIsEmpty

Purpose

Determine if the system user code buffer is empty (i.e., . contains only white space).

Syntax

```
%<LibSystemUserCodeIsEmpty(system, function, location)>
```

Arguments

`system`
Reference to a system.

`function`
Name of function to check (`Initialize`, `Start`, `Terminate`, `Output`, `Update`, `Derivative`, `Enable`, `Disable`, or `OutputUpdate`).

`location`
Location in which buffer is placed (`header`, `body`, `trailer`)

Returns

1 if the code buffer is empty or contains only white space; otherwise 0.

Description

`LSystemUserCodeIsEmpty` returns 1 if the code buffer is empty or contains only white space; otherwise it returns 0.

Purpose

Returns model task identifier (`TID`) and sets the global flag `CompiledModel.NeedTID` to 1.

Syntax

```
%<LibTID()>
```

Arguments

None.

Returns

`TID`.

Description

`LibTID` returns TID and informs the code generator that the TID is used in the context of the call to this function.

You should use this function when you want to get the `TID` instead of hard-coding `%<tTID>`.

A

model.rtw

Model.rtw File Contents

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

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

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

You need to understand the basic format of the *model*.rtw file if you are writing a TLC file for an S-function (i.e., inlining the S-function). You do not, however, need to know all the details about the *model*.rtw file. For the purpose of inlining an S-function, you only need to understand the concepts of the *model*.rtw file and how to access the information using the Target Language Compiler. Items such as signal connectivity and obtaining input and output connections for your S-function are contained within the *model*.rtw file using mapping tables. Processing this information directly in the Target Language Compiler is difficult. To simplify writing TLC files for S-functions, many library functions (which start with the prefix Lib) are provided. For example to access your inputs to your S-function, you should use LibBlockInputSignal.

When the Target Language Compiler calls the various functions that exist in your TLC file, the Block record for your S-function will be scoped. In this case, you have access to the Parameters and ParamSettings records shown on page A-64. If your S-function has an mdlRTW method, then you can control several fields within the Block record. For example, you can replace the Parameter record with more appropriately named records (which also can be data typed) using the ssWriteRTWParameters function. In mdlRTW, if you use the function ssWriteRTWParamSettings, then the Target Language Compiler will create an SFcnParameterSettings record in the Block record for your S-function. There are several other functions available to mdlRTW for adding information to the

model.rtw file. See *matlabroot*/simulink/src/sfuntmpl.doc for more information.

In addition, there are many Target Language Compiler library functions available to help you inline S-functions. See Chapter 5, "Target Language Compiler Function Library Reference," for a complete list of Target Language Compiler library functions.

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

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

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

```
SigLabel "velocity"
```

which specifies that the field (or variable) SigLabel contains the value "velocity". You can place this field in a record named Signal via

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

You can access fields within a record using the dot operator. For example, Signal.SignalLabel accesses the signal label field of the Signal record. You can change the local scope to any record in the Target Language Compiler using the with directive. This allows for both relative and absolute scoping. The Target Language Compiler first checks for the item being accessed in the local scope; if the item is not there, it then searches the global name pool (global scope).

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

```
NumSignals 2
Signal {
  SigLabel "velocity"
}
Signal {
  SigLabel "position"
}
```

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

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

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

An access to the field `Signal.ComplexSignal` returns `no`. The way the Target Language Compiler works is to first check the `Signal` record for the field (parameter) `ComplexSignal`. In this example, it does not exist, so the Target Language Compiler searches for the field `SignalDefaults.ComplexSignal` which has the value `no`. (If `SignalDefaults.ComplexSignal` did not exist, it generates an error.)

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

For example,

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

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

General Information and Solver Specification

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

In version 3.0 of Real-Time Workshop, the *model*.rtw file includes fields for variable step solvers and solver parameters. However, these fields are provided for use by future versions of Real-Time Workshop. Version 3.0 does not generate C code for use with variable step solvers.

The table below lists and describes the contents of the *model*.rtw file.

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

Variable/Record Name	Description
Name	Name of the Simulink model from which this *model*.rtw file was generated.
Version	Version of the *model*.rtw file.
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.
SolverMode	Either SingleTasking or MultiTasking.
FixedStep	Step size to be used.
TID01EQ	Either 0 or 1 indicating if the first two sample times are equal (1 if they are equal). This occurs where there is a continuous sample time and one or more discrete sample times and the fixed-step size is equal to the fastest discrete sample time.
}	

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

Variable/Record Name	Description
`VariableStepOpts {`	Only written if `SolverType` is `VariableStep`. Note that although `VariableStep` options are defined for *`model`*`.rtw`, the Real-Time Workshop does not currently support use of variable step solvers. You will see several references to fields for the variable step solvers.
`RelTol`	Relative tolerance.
`AbsTol`	Absolute tolerance.
`Refine`	Refine factor.
`MaxStep`	Maximum step size.
`InitialStep`	Initial step size.
`MaxOrder`	Maximum order for `ode15s`.
`}`	

Data Logging Information

The Workspace I/O page of the **Simulation Parameters** dialog box gives you the option of selecting whether to log data after executing model code generated from Real-Time Workshop. If you choose to log data, you must select one or more check boxes for Time, States, Output, or Final state. You can use the default variable names shown in the dialog box or replace these with your own name selections. For data logging, in order to conserve on memory and/or file space, you can limit data collection to the final N-points of your simulation. This is done by selecting the check box **Limit rows to last** and using the default value `1000`. Using the default setting saves the selected workspace variables to a buffer of length `1000`. If desired, you can provide another value for the total number of points per variable to store. You can also select to store the variables in one of three formats:

- Structure with time
- Structure
- Matrix

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

The table below lists and describes the data logging information for *`model`*`.rtw`.

Table A-2: Model.rtw Data Logging Information

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

Table A-2: Model.rtw Data Logging Information (Continued)

Variable/Record Name	Description
`StateSaveName`	Name of state variable or `""` if not being logged.
`OutputSaveName`	Name of output variable or `""` if not being logged.
`FinalStateName`	Name of final state variable or `""` if not being logged.
`StateSigSrc`	Only written if `SaveFormat` is not `Matrix` and either the states or the final states are being logged. This is an N-by-3 matrix with rows: `[sysIdx, blkIdx, blkStateIdx]` giving the location of the signal to be logged as a state. `sysIdx` and `blkIdx` give the source (i.e., a `Block` record), which specifies the states that are logged. The `blkStateIdx` is used to identify which part of the block the state is coming from. The `blkStateIdx` will be: • -2 if the logged signal is the continuous state vector • -1 if the logged signal is the discrete state vector • >= 0 implies the logged signal is the `blkStateIdxD Work` (data type work vector) of `blkIdx` block in `sysIdx` system Note that `sysIdx` and `blkIdx` are valid even if `blkStateIdx < 0`, and `N` is the number of signals being logged as states.
}	

Data Structure Sizes

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

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

The following table describes the data structure written to *model*.rtw.

Table A-3: Model.rtw Model Data Structure Sizes

Variable/Record Name	Description
NumModelInputs	Sum of all root-level import block widths. This is the length of the external input vector, U.
NumModelOutputs	Sum of all root-level outport block widths. This is the length of the external output vector, Y.
NumNonVirtBlocksInModel	Total number of nonvirtual blocks in the model.
DirectFeedthrough	Does 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.

Table A-3: Model.rtw Model Data Structure Sizes (Continued)

Variable/Record Name	Description
`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. There are two vectors of this length: the zero-crossing signals (`nonsampledZCs`), and the zero-crossing directions (`nonsampledZCdirs`). Nonsampled zero-crossings are derived from continuous signals that have a discontinuity in their first derivative. Nonsampled zero-crossings only exist for variable step solvers. The Abs block is an example of a block that has an intrinsic, nonsampled zero-crossing to detect when its input crosses zero.
`NumZCEvents`	Length of the model zero-crossing event vector (`zcEvents`).
`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.
`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.
`NumDWork`	Total number of data type work vector elements. This is the sum of the widths of all data type work vectors in the model..

Table A-3: Model.rtw Model Data Structure Sizes (Continued)

Variable/Record Name	Description
`NumDWorkRecords`	Total number of data type work records in the model. Each record contains a vector as well as information describing the vector. For example the model may contain two data type work records where one record contains a vector of length 3 and the other contains a vector of length 9 where each vector is of a different data type. In this case `NumDWorkRecords` is 2.
`NumDataStoreElements`	Total number of data store elements. This is the sum of the widths of all data store memory blocks in your model.
`NumBlockSignals`	Sum of the widths of all output ports of all nonvirtual blocks in the model. This is the length of the block I/O vector, `blockIO`.
`NumBlockParams`	Number of modifiable parameter elements (`params`). For example, the Gain block parameter contains modifiable parameter elements.
`NumAlgebraicLoops`	Number of algebraic loops in the model.

Sample Time Information

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

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

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

Table A-4: Model.rtw Sample Times

Variable/Record Name	Description
AllSampleTimesInherited	yes if all blocks in the model have inherited sample times, no otherwise.
InvariantConstants	yes if invariant constants (i.e., Inline parameters check box) is on, no if invariant constants is off.
FundamentalStepSize	Fundamental step size or 0.0 if one cannot be determined. Fixed step solvers will always have a nonzero step size. Variable step solvers may have a fundamental step size of 0.0 if one can not be computed from the sample times in the model.
NumSampleTimes	Number of sample times in the model followed by SampleTime info records, giving the TID (task ID), an index into the sample time table, and the period and offset for the sample time.
SampleTime {	One record for each sample time.
TID	Task ID for this sample time.
PeriodAndOffset	Period and offset for this sample time.
}	

Data Type Information

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

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

Table A-5: Model.rtw Data Types

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

Block Type Counts

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

Table A-6: Model.rtw Block Type Counts

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

Model Signals and Subsystems

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

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

Table A-7: Model.rtw Model Signals and Subsystems

Signals and Subsytems	Description
`Root Signals {`	Signal and block information in the root window.
`ChildSubsystemIndices`	Vector of integers specifying the subsystems that are directly contained within the root system. The indices index into the `CompiledModel.Subsystem` record list.
`NumSignals`	Number of block output signals (including virtual) blocks.
`Signal {`	One record for each block output signal (i.e., length of this list is `NumSignals`).
`Block`	`[sysIdx, blockIdx]` or block name string if a virtual block.
`SigLabel`	Signal label if present.
`OutputPort`	`[outputPortIndex, outputPortWidth]`.
`DataTypeIdx`	Index into the `CompiledModel.DataTypes.DataType` record list. Only written for nonvirtual blocks and if data type index is not 0 (the default data type of 0 corresponds to `real_T`).
`ComplexSignal`	`yes/no`: is the signal complex? Only written for nonvirtual blocks and if signal is complex.
`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.

Table A-7: Model.rtw Model Signals and Subsystems (Continued)

`BlockMap`	Vector of length `NumBlocks` giving the `blockIdx` for each nonvirtual block in the subsystem.
`}`	
`NumVirtualSubsystems`	Total number of virtual (non-empty) 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 complete mapping to the blocks full path name (e.g. `<s2/gain1>`).

`SLName`	Unmodified Simulink name. This is only written if it is *not* equal to `Name`. This will occur when generating code using the `rtwgen StringMapings` argument. For the Real-Time Workshop C targets, any block name which contains a new-line, `'/*'`, or `'*/'` will have these characters re-mapped. For example suppose the Simulink block name is:
	`my block name`
	`/* comment */`
	The *`model`*`.rtw` file will contain:
	`Name "<Root>/my block name //+ comment +//"`
	`SLName "<Root>/my block name\n//* comment *//"`

`Virtual`	`yes/no`: Whether or not the subsystem is virtual.
`ChildSubsystemIndices`	Vector of integers specifying the subsystems that are directly contained within this subsystem. The indices index into the `CompiledModel.Subsystem` record list.
`NumSignals`	Number of block output signals (including virtual) blocks.
`Signal {`	One record for each block output signal (i.e., length of this list is `NumSignals`).
`Block`	`[sysIdx, blockIdx]` or block name string if a virtual block.
`OutportName`	This field is written only if the signal is emanating from a subsystem. It is the Outport block name corresponding to the output signal of a subsystem block.
`SigLabel`	Signal label if present.
`OutputPort`	`[outputPortIndex, outputPortWidth]`.

Table A-7: Model.rtw Model Signals and Subsystems (Continued)

`DataTypeIdx`	Index into the `CompiledModel.DataTypes.DataType` record list. Only written for nonvirtual blocks and if data type index is not 0 (i.e., `real_T`).
`ComplexSignal`	`yes`: Only written for nonvirtual blocks and if signal is complex.
`SignalSrc`	Vector of length `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.
`}`	

External Inputs and Outputs

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

Table A-8: Model.rtw External Inputs and Outputs

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

Table A-8: Model.rtw External Inputs and Outputs (Continued)

`SigLabel`	Signal label entered by user.
`DirectFeedThrough`	Only written if this inport doesn't require it's signal when `MdlOutputs` is called.
`StorageClass`	Only written if not `Auto`. This setting determines how this signal is declared.
`StorageTypeQualifier`	Only written if not empty.
`}`	
`}`	
`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.
`ExternalOutputs {`	External outputs (root outports) from the block diagram
`NumExternalOutputs`	Number of ExternalOutput records that follow. This is equal to the number of root level outports.
`ExternalOutput {`	One record per root-level outport block.
`Block`	`[sysIdx, blockIdx]` of the outport block.
`SigIdx`	`[externalOutputVectorIndex, signalWidth]`
`SigLabel`	Label on the input port signal, if any.
`}`	
`}`	

Data Store Information

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

Table A-9: Model.rtw Data Store Information

Variable/Record Name	Description
DataStoreDefaults {	Defaults for the data store records.
DataTypeIdx	0: Default is real_T.
ComplexSignal	no: Default is not complex.
}	
DataStores {	Record giving the data stores found in the block diagram.
NumDataStores	Number of data stores in the block diagram.
DataStore {	One record for each data store.
Name	Name of block declaring the data store.
SLName	Unmodified Simulink name. This is only written if it is *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.
}	
}	

Block I/O Information

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

The `BlockOutputs` record provides a listing of all blocks that output signals external to the block output vector. Several optimizations that affect block outputs are provided through Simulink dialog boxes. The Diagnostics page provides a check box *Disable optimized block I/O storage*. When you select this option, the Target Language Compiler places a record for each block in the *`model`*`.rtw` file. If you do not choose this option, you can selectively add the output from a particular block by specifying the block output as a test point. To specify a block output as a test point, select a line and then select edit->signal properties->check box Displayable (Test Point). Once you have tagged a signal as a test point, the Target Language Compiler always writes to the block I/O vector. If a signal is not visible in the block outputs vector, it will allow reuse of its memory location by several blocks. This can substantially reduce memory requirements.

Table A-10: Model.rtw Block I/O Information

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

Table A-10: Model.rtw Block I/O Information (Continued)

`Invariant`	`no`: The default is that this signal has a non-constant sample time and change during execution.
`InitialValue`	`[]`: The default initial value is empty for non-invariant signals.
`DataTypeIdx`	`0`: The default data type is `real_T`.
`ComplexSignal`	`no`: The default is a non-complex real valued signal.
`SigSrc`	`[]`: The default source is nonexistent for case of multiple sources which is created via reused signals.
`SigLabel`	`""`: No signal label on the line.
`SigConnected`	`all`: All destination elements of the signal are connected to other nonvirtual blocks or root outports.
`}`	
`ReusedBlockOutputDefaults {`	
`SigLabel`	`""`: No signal label on the line.
`SigConnected`	`all`: All destination elements of signal are connected to other nonvirtual blocks or root outports.
`}`	
`MergedBlockOutputDefaults {`	
`SigLabel`	`""`: No signal label on the line.
`SigConnected`	`all`: All destination elements of signal are connected to other nonvirtual blocks or root outports.
`}`	
`NumBlockOutputs`	Number of data output port signals.
`BlockOutput {`	One record for each data output signal.
`Identifier`	Unique variable name across all block outputs.
`SigIdx`	`[blockIOVectorIndex, signalWidth]`.

Table A-10: Model.rtw Block I/O Information (Continued)

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

Table A-10: Model.rtw Block I/O Information (Continued)

`Identifer`	Unique variable name across all block outputs.
`SigSrc`	`[systemIndex, blockIndex, outputPortIndex]`.
`SigLabel`	Signal label entered by user. Only written if non-empty (`""`).
`}`	
`ReusedBlockOutput {`	Only written when this BlockOutput record is being reused by multiple blocks. There is one record for each block output port that is reused by this BlockOutput record.
`Identifer`	Unique variable name across all block outputs.
`SigSrc`	`[systemIndex, blockIndex, outputPortIndex]`.
`SigLabel`	Signal label entered by user. Only written if non-empty (`""`).
`MergedBlockOutput {`	Only written if this `ReusedBlockOutput` record corresponds to a Merge block. In this case, the number of `MergedBlockOutput` records is equal to the number of input ports on the Merge block. See above for contents of the `MergedBlockOutput` records.
`}`	
`}`	
`}`	
`}`	
`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`.

Data Type Work (DWork) Information

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

Table A-11: Model.rtw Data Type (DWork) Information

Variable/Record Name	Description
`DWorkRecords {`	List of all data type work vectors in the model. There is one `DWorkRecord` record for each data type work vector in the model. The *source* of a data type work vector is a block. A block can have zero or more data type work vectors.
`DWorkRecordDefaults {`	Default values for the following `DWorkRecord` records.
`DataTypeIdx`	`0`: Default vector of a `DWorkRecord` record is a `real_T` vector.
`ComplexSignal`	`no`: Default vector of a `DWorkRecord` record is a non-complex vector.
`UsedAsDState`	`no`: Default vector of a `DWorkRecord` record is not logged as a state (i.e., doesn't go in to the *`model`*`.mat` file).
`}`	
`NumDWorkRecords`	Number of data type work vectors in the model. Each block can register 0 or more data type work vectors.
`DWorkRecord {`	One `DWorkRecord` record for each data type work vector.
`BlockIdentifier`	Unique identifier across all blocks indicating which block this data type work vector is from.
`Name`	Name of this data type work vector from the block's perspective.
`Width`	Length of the data type work vector.
`DataTypeIdx`	Index into the `CompiledModel.DataTypes.DataType` list. Only written if data type is a non-real_T (i.e., not a 0).

Table A-11: Model.rtw Data Type (DWork) Information (Continued)

Variable/Record Name	Description
`ComplexSignal`	`yes`: Only written if this data type work vector is complex.
`UsedAsDState`	`yes`: Only written if this data type work vector is logged when the simulation parameters workspace I/O state logging is selected.
`DWorkSrc`	`[systemIdx, blockIdex, dworkIdx]`
`}`	
`}`	

State Mapping Information

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

Table A-12: Model.rtw State Mapping Information

Variable/Record Name	Description
`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`.

Block Record Defaults

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

Table A-13: Model.rtw Block Defaults

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

Parameter Record Defaults

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

Table A-14: Model.rtw Parameter Defaults

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

Data and Control Input Port Defaults

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

Table A-15: Model.rtw Data and Control Input Port Defaults

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

System Record

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

Table A-16: Model.rtw System Record

Variable/Record Name	Description
`System {`	One for each system in the model. This is equal to `NumNonvirtSubsystems` plus 1 for the root system.
`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.

Table A-16: Model.rtw System Record (Continued)

Variable/Record Name	Description
`ZeroCrossFcn`	Name of nonsampled zero-crossing function for `enable` systems using variable step solver.
`OutputUpdateFcn`	Name of output/update function for `trigger` or `enable_with_trigger` systems.
`NumBlocks`	Number of nonvirtual blocks in the system.
`BlocksIdx`	0: Location of the first nonvirtual block in the following `Block` record list.
`NumVirtualOutportBlocks`	For the root system, the number of virtual outport blocks is 0 (since all root outport blocks are nonvirtual). For a system corresponding to a conditionally executed subsystem, this is equal to the number of outport blocks in the subsystem. For each of these virtual outport blocks, there is a corresponding `Block` record which appears after all the nonvirtual `Block` records.
`VirtualOutportBlockIdx`	Starting index in the following `Block` record list of the virtual outport blocks.
`NumTotalBlocks`	Number of blocks in the system (sum of `NumBlocks` and `NumVirtualOutportBlocks`).
`Block {`	One for each nonvirtual block in the system. The virtual outport block records are described below.
`Type`	Block type, 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 resides in a masked subsystem.
`Tag`	This is the text that can be attached to a block via the command: `set_param('block','Tag','text')` This parameter is written if the text is non-empty.

Table A-16: Model.rtw System Record (Continued)

Variable/Record Name	Description

`RTWdata {`	The `RTWdata` general record is only written if the RTWdata property of a block is non-empty. The RTWdata is created using the command:
	`set_param('block','RTWData',val)`
	where `val` is a Matlab struct of string. For example:
	`val.field1 = 'field1 value'`
	`val.field2 = 'field2 value'`

`field1`	`"field1 value"`
`feild2`	`"field2 value"`
`}`	
`Name`	Block name preceded with a `<root>` or `<S#>` token.
`SLName`	Unmodified Simulink name. This is only written if it is *not* equal to `Name`.
`Identifier`	Unique identifer across all blocks in the model.
`PortBasedSampleTimes`	`yes`. Only written if block specified port based sample times.
`InputPortTIDs`	Only written if port sample time information is available.
`OutputPortTIDs`	Only written if port sample time information is available.
`TID`	Task ID, which can be one of: • Integer >= 0, giving the index into the sample time table. • Vector of two or more elements indicating that this block has multiple sample times. • `constant` indicating that the block is constant and doesn't have a task ID. • `triggered` indicating that the block is triggered and doesn't have a task ID. • `Subsystem` indicating that this block is a conditionally executed subsystem and the TID transitions are to be handled by the corresponding system.

Table A-16: Model.rtw System Record (Continued)

Variable/Record Name	Description
`SubsystemTID`	Only written if TID equals `Subsystem`. This is the actual value of the subsystem TID (i.e., `integer`, `vector`, `constant`, or `triggered`).
`FundamentalTID`	Only written for multirate or hybrid enabled subsystems. This gives the sample time as the greatest common divisor of all sample times in the system.
`SampleTimeIdx`	Actual sample time of block. Only written for zero order hold and unit delay blocks.
`AlgebraicLoopId`	This ID identifies what algebraic loop this block is in. If this field is not present, the ID is 0 and the block is not part of an algebraic loop.
`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`.
`NumDWork`	Number of `DWork` records block has declared. There is one `DWork` record for each data type work vector of the block.
`DWork {`	One record for each data type work vector.
`Name`	Name of the data type work vector.
`RecordIdx`	Index of this record in the model wide `CompiledModel.DWorkRecords.DWorkRecord` list.
`}`	
`ModeVector`	Specified as `[N,I]` where `N` is the number of modes and `I` is the index into `modeVect`. Not present if `N==0`.

Table A-16: Model.rtw System Record (Continued)

Variable/Record Name	Description
`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.
`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.
`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.
`DataTypeIdx`	Index into the `CompiledModel.DataTypes.DataType` record list giving the data type of this port. Only written if not 0 (see `CompiledModel.DataInputPortDefaults.DataTypeIdx`).
`ComplexSignal`	Is this port complex? Only written if `yes`. The default from `CompiledModel.DataInputPortDefaults.ComplexSignal` is `no`.

Table A-16: Model.rtw System Record (Continued)

Variable/Record Name	Description
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), unconnected ground (GO), or FcnCall indicating the source is a function-call.
RollRegions	A vector (e.g., [1:5, 6:10, 11]) giving the contiguous regions for this data input port over which *for* loops can be used. This is always written for S-function blocks, otherwise it is written only if it is different from the block RollRegions.
DirectFeedThrough	Does this input port have direct feedthrough? Only written if WriteBlockConnections is on and the value this port does not have direct feedthrough, in which case no is written.
BufferDstPort	Only written if this input port is used by an output port of this block. The default is CompiledModel.DataInputPortDefaults.BufferDstPort which is -1.
}	
NumControlPorts	Number of control (e.g., trigger or enable) input ports. Only written if nonzero.
ControlPort {	One record for control input port.
Type	Type of control port: enable, trigger, or function-call.
Width	Width (i.e. vector length) of the signal entering this input port.
DataTypeIdx	Index into the CompiledModel.DataTypes.DataType record list giving the data type of this port. Only written if not 0 (see CompiledModel.ControlInputPortDefaults.DataTypeIdx).
ComplexSignal	Is this port complex? Only written if yes. The default from CompiledModel.ControlInputPortDefaults.ComplexSignal is no.
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 (GO).

Table A-16: Model.rtw System Record (Continued)

Variable/Record Name	Description
SignalSrcTID	Vector of length `Width` giving the TID as an integer index, `trigger`, or `constant` identifier for each signal entering this control port. This is the rate at which the signal is entering this port. If the subsystem block has a triggered sample time, then the signal source must be triggered.
RollRegions	A vector (e.g., [1:5, 6:10, 11]) giving the contiguous regions for this data input port over which *for* loops can be used. This is always written for S-function blocks, otherwise it is written only if it is different from the block `RollRegions`.
DirectFeedThrough	Does this input port have direct feedthrough? Only written if `WriteBlockConnections` is on and the value this port does not have direct feedthrough, in which case `no` is written.
BufferDstPort	Only written if this input port is used by an output port of this block.The default is `CompiledModel.ControlInputPortDefaults.BufferDstPort` which is `-1`.
}	
NumDataOutputPorts	Number of output ports. Only written if nonzero.
DataOutputPortIndices	Indices into `BlockOutputs` record. Only written if `NumDataOutputPorts > 0`.
Connections {	Only written if this is an S-function block, or the `WriteBlockConnections` rtwgen options was specified as on.
InputPortContiguous	Vector of length `NumDataInputPorts` containing `yes`, `no`, or `grounded`.
DirectSrcConn	Vector of length `NumDataInputPorts` containing yes or no as to whether or not the input port is directly connected to a nonvirtual source block.
DirectDstConn	Vector of length `NumDataOutputPorts` containing `yes` or `no` as to whether or not the output port is directly connected to a signal nonvirtual destination block.
DataOutputPort {	One record for each data output port.

Table A-16: Model.rtw System Record (Continued)

Variable/Record Name	Description
NumConnPoints	Number of destination *connection points*. A destination connection point is defined to be a one-to-one connection with elements from the output (src) port to the destination block and port.
ConnPoint {	
SrcSignal	Vector of length two giving the range of signal elements for the connection: `[startIdx, length]` Where `startIdx` is the starting index of the connection in the output port and length is the number of elements in the connection.
DstBlockAndPortEl	Vector of length four giving the destination connection: `[sysIdx, blkIdx, inputPortIdx, inputPortEl]` `sysIdx` is the index of the system record. `blkIdx` is the index with in system record of the destination block. `inputPortIdx` is the index of the destination input port. `inputPortEl` is the starting offset within the port of the connection.
}	
}	
}	
Parameters	Specified as `[N,M]` where `N` is the number of `Parameter` records that follow, `M` is the number of modifiable parameter elements. Not present if `N==0`.
Parameter {	One record for each parameter.
Name	Name of the parameter as defined by the block.
Value	Value of the parameter.
DataTypeIdx	Data type index of the parameter into the `CompiledModel.DataTypes.DataType` records. Only written if not 0 (i.e., not real_T).

Table A-16: Model.rtw System Record (Continued)

Variable/Record Name	Description
`ComplexSignal`	Is this parameter complex? Only written if `yes`.
`String`	String entered in the Simulink block dialog box.
`StringType`	One of: • `"Computed"` indicating the parameter is computed from values entered in the Simulink dialog box. • `"Variable"` indicating the parameter is derived from a single MATLAB variable. • `"Expression"` indicating the parameter is a MATLAB expression.
`ASTNode {`	Currently this is only one level deep for and contains the direct mapping of this parameter to the model parameters record list.
`Op`	`ModelParameterIdx` - indicates that this node is to be used when indexing the model parameters list.
`Value`	Index into the `CompiledModel.ModelParameters.Parameter` list.
`}`	
`}`	
`ParamSettings {`	Optional record specific to block.
`blockSpecificName`	Block specific settings.
`}`	
`}`	
`<S-function fields>`	Optional fields (parameters and/or records) that are written to the *`model`*`.rtw` file by the your specific S-function `mdlRTW` method.
`Block {`	One block record (after the nonvritual block records) for each virtual outport block in the system.
`Type`	Outport
`Name`	Block name preceded with a `<root>` or `<S#>` token.

Table A-16: Model.rtw System Record (Continued)

Variable/Record Name	Description
`SLName`	Unmodified Simulink name. This is only written if it is *not* equal to `Name`.
`Identifier`	Unique identifer across all blocks.
`RollRegions`	A vector (e.g., [1:5, 6:10, 11]) giving the contiguous regions over which *for* loops can be used.
`NumDataInputPorts`	1
`DataInputPort {`	See nonvirtual block `DataInputPort` record.
`}`	
`}`	
`EmptySubsysInfo {`	
`NumRTWdatas`	Number of empty subsystem blocks which have `set_param(block, 'RTWdata', val)` specified where val is a struct of strings.

`RTWdata {`	The `RTWdata` general record is only written if the RTWdata property of a block is non-empty. The RTWdata is created using the command:
	`set_param('block','RTWData',val)`
	where val is a Matlab struct of string. For example:
	`val.field1 = 'field1 value'`
	`val.field2 = 'field2 value'`

`field1`	`"field1 value"`
`field2`	`"field2 value"`
`}`	
`}`	

Model Parameters Record

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

Table A-17: Model.rtw Model Parameters Record

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

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

Variable/Record Name	Description
`Tunable`	If inlined parameters check box is off, then all parameter values are tunable (they will reside in the `rtP` vector). If inlined is on, then tunable means that this parameter has been selectively non-inlined. It will be placed in memory according to the Storage class. Note that the default value is `'no'` (in which case this field is not written).
`StorageClass`	Specifies where to declare/place this parameter value in memory (`Auto`, `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`). Default value is `Auto` in which case this field is not written to the *`model`*`.rtw` file.
`TypeQualifier`	String used as a type qualifier for the declaration of the parameter (e.g., `"static"`)
`ReferencedBy`	An Nx3 matrix. Each row specifies a system, block, parameter index triplet that identifies a usage of this parameter value. If N>1, then this parameter value is shared by multiple blocks.
`  }`	
`}`	

Model Checksums

Checksums are created which are unique for each model.

Table A-18: Model.rtw Checksums

Variable/Record Name	Description
`BlockParamChecksum`	This is a hash-based checksum for the block parameter values and identifier names.
`ModelChecksum`	This is a hash-based checksum for the model structure.

Common Fields of Block Parameter Records

Each block may have parameters. All parameters are written out to the *model*.rtw file in Parameter records which are contained with the Block records. There is one Parameter record for each block parameter (i.e. Block.Parameter[*i*]). A parameter in this context only refers to ones that external mode can tune. Therefore there may not be a one-to-one mapping between parameters in the *model*.rtw file and the parameter dialog for the block. The table below describes the common attributes for each Parameter record. The following Parameter record fields are only present if the field values differ from the defaults given in "Model.rtw Parameter Defaults" on page A-30.

Table A-19: Common Fields of Block Parameter Records

Variable/Record Name	Description
DataTypeIdx	Index giving the corresponding DataType record (sub record of CompiledModel.DataTypes.
ComplexSignal	yes/no: Is this parameter complex?

Block Specific Records

Each Simulink built-in block has an associated block record, which covers all possible configurations of that block. The following table provides a complete listing of built-in blocks and their associated records. The blocks are listed in alphabetical order. The Target Language Compiler also has an associated TLC file for each block that specifies how code is generated for each block.

The following table describes the block specific records written for the Simulink blocks (excluding the `DataTypeIdx`, and `ComplexSignal` fields)

Table A-20: Model.rtw Block Specific Records

Block Type: `AbsoluteValue` - No block specific records

Block Type: `Backlash` (example with a backlash width of 2.08 and and 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: `BusSelector`

No block specific records.

Block Type: `Clock`

No block specific records.

Table A-20: Model.rtw 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: `ComplexToMagnitudeAngle`

```
ParamSettings {
  Output                  One of "Magnitude", "Angle", or "MagnitudeAndAngle"
                          indicating what the output port(s) are producing.
}
```

Block Type: `ComplexToRealImag`

```
ParamSettings {
  Output                  One of "Real", "Imag", or "RealAndImag" indicating what the
                          output port(s) are producing.
}
```

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

```
Parameter {
  Name                    "Value"
  Value                   [1.0, 2.0, 3.0, 4.0, 5.0]
  String                  "1:5"
  StringType              "Expression"
}
```

Block Type: `DataStoreMemory`

Virtual. Not written to RTW file.

Table A-20: Model.rtw Block Specific Records (Continued)

Block Type: `DataStoreRead`

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

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-20: Model.rtw 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 (`RWork`). 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
}
```

Block Type: `DigitalClock`
No block specific records.

Block Type: `DiscreteFilter`
See Model.rtw Linear Block Specific Records on page A-75.

Table A-20: Model.rtw Block Specific Records (Continued)

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`	`[0.0]`
`  String`	`"0"`
`  StringType`	`"Expression"`
`}`	
`Parameter {`	
`  Name`	`"UpperSaturationLimit"`
`  Value`	`[0.75]`
`  String`	`".75"`
`  StringType`	`"Expression"`
`}`	
`Parameter {`	
`  Name`	`"LowerSaturationLimit"`
`  Value`	`[-0.25, 0.0, -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`

Table A-20: Model.rtw Block Specific Records (Continued)

InitialConditionSource	internal or external
LimitOutput	on or off
ShowSaturationPort	on or off
ShowStatePort	on or off
ExternalXO	Only written when initial condition (IC) source is external. This is the initial value of the signal entering the IC port.

```
}
```

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                   [2.0]
  String                  "2"
  StringType              "Expression"
}
Parameter {
  Name                    "PulseWidth"
  Value                   [1.0]
  String                  "1"
  StringType              "Expression"
}
ParamSettings {
  PhaseDelay              [0.0]
}
NumIWorkDefines           1   (There is one integer work for each output signal.)
IWorkDefine {
```

Table A-20: Model.rtw Block Specific Records (Continued)

```
  Name                    "ClockTicksCounter"
  Width                   1
}
```

Block Type: `DiscreteStateSpace`
See Model.rtw Linear Block Specific Records on page A-75.

Block Type: `DiscreteTransferFcn`
See Model.rtw Linear Block Specific Records on page A-75.

Block Type: `DiscreteZeroPole`
See Model.rtw Linear Block Specific Records on page A-75.

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

Block Type: `From`
Virtual. Not written to `model.rtw` file.

Block Type: `FromFile`

```
ParamSettings {
 FileName                 Name of MAT-file to read data from.
  NumPoints               Number of data points (integer).
  TUdata                  Real data from the MAT-file. Not present if using the rapid
                          simulation target, rsim.
```

Table A-20: Model.rtw Block Specific Records (Continued)

Width	Number of columns in MAT-file TU block data.
}	

Block Type: `FromWorkspace`

`ParamSettings {`	
VariableName	Name of variable in `Data` field in block parameter dialog box
DataFormat	`"Matrix"` or `"Structure"`
Interpolate	Interpolate flag is `on`/`off` (see the From Workspace block description in the "Using Simulink" manual).
HoldFinalValue	Hold final value flag is `on`/`off` (see the From Workspace block description in the "Using Simulink" manual).
`NumPoints`	Number of data points (rows) over which to read values from as time moves forward and write to the output port.
TimePresent	Tells if the `Time` vector is present. The rapid simulation target does not need the time vector.
Time	Time vector. May or may not be present. If data format is Matrix, then this field is always present. If data format is Struct then this field is present only if the time field exists.
`Data`	Data from the workspace variable(s).Data to put on the output port. Not present if using the rapid simulation target, rsim.
}	

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                      "+"
  LHS {
    Op                    "SIN"
    LHS {
      Op                  "U"
```

Table A-20: Model.rtw Block Specific Records (Continued)

```
        LHS {
          Op                  "NUM"
          Value               1.0 (note for Op, "U", this must be casted to a Number)
        }
      }
    }
    RHS {
      Op                      "NUM"
      Value                   10.0
    }
  }
```

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

```
  Parameter {
    Name                      "Gain"
    Value                     [1.0, 2.0, 3.0, 4.0, 5.0]
    String                    "1:5"
    StringType                "Expression"
  }
```

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                     [0.0]
    String                    "0"
    StringType                "Expression"
  }
```

Table A-20: Model.rtw Block Specific Records (Continued)

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                   [1.0, 2.0, 3.0, 4.0, 5.0]
}
```

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                   [0.0]
  String                  "0"
  StringType              "Expression"
}
Parameter {
  Name                    "UpperSaturationLimit"
  Value                   [0.75]
  String                  ".75"
  StringType              "Expression"
}
Parameter {
  Name                    "LowerSaturationLimit"
  Value                   [-0.25, 0.0, -0.75]
  String                  "[-.25 0 -.75]"
  StringType              "Expression"
}
```

Table A-20: Model.rtw 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`, corresponding to the function `y = f(x)`.
`Name`	`"InputValues"`
`Value`	`[–5.0, –4.0, –3.0, –2.0, –1.0, 0.0]`
`String`	`"[–5:0]"`
`StringType`	`"Expression"`
`}`	
Parameter {	The output values, `y`, of the function `y = f(x)`.
`Name`	`"OutputValues"`
`Value`	`[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]`
`String`	`"[0:5]"`
`StringType`	`"Expression"`
`}`	
`Parameter {`	Slope is computed as `(y(i+1)–y(i))/(x(i+1)–x(i))`
`Name`	`"Slopes"`
`Value`	`[1.0, 1.0, 1.0, 1.0, 1.0, 0.0]`

Table A-20: Model.rtw Block Specific Records (Continued)

String	""
StringType	"Computed"
}	
Parameter {	The output value of the block when the input to the block is zero. In this example 5.0.
Name	"OutputAtZero"
Value	[5.0]
String	""
StringType	"Computed"
}	
ParamSettings {	
ZeroTechnique	The type of lookup being performed. This doesn't change during model execution. The possibilities are: `NormalInterp`, `AverageValue`, or `Middle_Value`.
}	

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)`.
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`, for the function `z = f(x,y)`.
Name	"OutputValues"
Value	Matrix(2,3)

Table A-20: Model.rtw Block Specific Records (Continued)

```
    [[4.0, 5.0, 6.0]; [16.0, 18.0, 20.0];]
    String                  "[[4, 5, 6]; [16, 18, 20]]"
    StringType              "Expression"
  }
  Parameter {               The output value of the block when the row input, x, is zero.
    Name                    "OutputAtRowZero"
    Value                   [–8.0, –8.0, –8.0]
    String                  ""
    StringType              "Computed"
  }
  ParamSettings {
    ColZeroTechnique        NormalInterp, AverageValue, or MiddleValue.
    ColZeroIndex            0
    RowZeroTechnique        NormalInterp, AverageValue, or MiddleValue.
  }
```

Block Type: MagnitudeAngleToComplex

```
  Parameter {               Only written when there is one input port.
    Name                    "ConstantPart"
    Value                   [1.570796326794897]
    String                  "pi/2"
    StringType              "Expression"
  }
  ParamSettings {
    Input                   One of "Magnitude", "Angle", or "MagnitudeAndAngle".
  }
```

Block Type: Math

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

Table A-20: Model.rtw Block Specific Records (Continued)

Block Type: `MATLABFcn`

There is no support for the MATLAB Fcn block in the Real-Time Workshop.

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

```
Parameter {
  Name                    "X0"
  Value                   [0.0]
  String                  "0"
  StringType              "Expression"
}
NumRWorkDefines           1
  Name                    "PrevU"
 Width                    2
}
```

Block Type: `Merge`

```
ParamSettings {
  InitialOutput           []
  DataTypeIdx             0
}
```

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`

Table A-20: Model.rtw Block Specific Records (Continued)

`ParamSettings {`	Nonvirtual (root-level) outport block settings:
`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`.
`}`	
`Parameter {`	Only written for virtual outport blocks at top of a nonvirtual subsystem.
`Name`	`"InitialOutput"`
`Value`	`[]`
`String`	`"[]"`
`StringType`	`"Expression"`
`}`	

Block Type: `Probe`

`ParamSettings {`	
`ProbeWidth`	`on` or `off`
`ProbeSampleTime`	`on` or `off`
`ProbeComplexSignal`	`on` or `off`
`}`	

Block Type: `Product`

No block specific records.

Block Type: `Quantizer` (example of a quantizer block with a quantization interval of 0.5).

`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 {`

Table A-20: Model.rtw Block Specific Records (Continued)

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

Table A-20: Model.rtw Block Specific Records (Continued)

`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"`
Value	`[2.220446049250313e–16]`
`String`	`"eps"`
`StringType`	`"Variable"`
`}`	
`Parameter {`	
`Name`	`"OffSwitchValue"`
`Value`	`[2.220446049250313e–16]`
`String`	`"eps"`
`StringType`	`"Variable"`

Table A-20: Model.rtw Block Specific Records (Continued)

```
    }
    Parameter {
      Name                    "OnOutputValue"
      Value                   [1]
      String                  "1"
      StringType              "Expression"
    }
    Parameter {
      Name                    "OffSwitchValue"
      Value                   [0.0]
      String                  "0"
      StringType              "Expression"
    }
```

Block Type: `ResetIntegrator`

```
    Parameter {
      Name                    "InitialCondition"
      Value                   [0.0]
      String                  "0"
    StringType                "Expression"
    }
```

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]
```

Table A-20: Model.rtw Block Specific Records (Continued)

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

Block Type: Scope

ParamSettings {	
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. These Parameter records are only present for S-functions which don't contain the mdlRTW method. See below for S-functions which contain the mdlRTW method.
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.

Table A-20: Model.rtw Block Specific Records (Continued)

String	""
StringType	"Computed"
}	
ParamSettings {	
FunctionName	Name of S-function.
SFunctionLevel	Level of the S-function (1 or 2).
FunctionType	Type of S-function: "M-File", "C-MEX", or "FORTRAN-MEX".
Inlined	yes or no. Does a *sfunctionname*.tlc file exist in the current directory, the directory in which *sfuntionname.mex* exists, or the tlc_*lang* subdirectory of where the *sfunctionname.mex* file resides?
DirectFeedthrough	For level 1 S-functions, this will be written as yes or no. For level 2 S-functions, this will be a vector of yes or no for each input port.
UsingUPtrs	yes/no: Is the Level 1 C MEX S-function using ssGetUPtrs or ssGetU? Level 2 S-functions can only access inputs through *uptrs* and this field will not be present.
InputContiguous	For level 1 S-functions, this will be yes or no. For level 2 S-functions this is a vector of yes or no for each input port.
SampleTimesToSet	M-by-2 matrix of sample time indices indicating any sample times specified by the S-function in mdlInitializeSizes and 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", "PWork", "DO",..., "Dn". For example ["U0", "U1", "Y0"]. For a level 1 S-function only U or Y will be used whereas for a level 2 S-function, U0, U1,..., Un, Y0, Y1,... Yn will be used. This includes dynamically typed vectors, i.e., data type and complex signals. For example, if U0 is in this list either width, data type or complex signal of U0 is dynamically sized (or typed).

Table A-20: Model.rtw Block Specific Records (Continued)

`SFcnmdRoutines`	Vector containing any of:
	`["mdlInitializeSizes",`
	`"mdlInitializeSampleTimes",`
	`"mdlInitializeConditions",`
	`"mdlStart",`
	`"mdlOutputs",`
	`"mdlUpdate",`
	`"mdlDerivatives",`
	`"mdlTerminate"`
	`"mdlRTW"]`
	Indicating which routines need to be executed. Only written for level 2 S-functions.

`}`	
`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.
`}`	

The `ParamSettings` record will be following by S-function specific information (`Parameters` and general S-function records which should start with `SFcn`) if there is a `mdlRTW` method.

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

```
Parameter {
  Name                  "Amplitude"
  Value                 [1.0]
  String                "1"
  StringType            "Expression"
}
Parameter {
  Name                  "Frequency"
  Value                 [1.0]
  String                "1"
```

Table A-20: Model.rtw Block Specific Records (Continued)

```
    StringType              "Expression"
  }
  ParamSettings {
    WaveForm                sine, square, or sawtooth
    TwoPi                   6.283185307179586
    StringType              "Expression"
  }
```

Block Type: `Signum`

No block specific records.

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                   [1.0]
    String                  "1"
    StringType              "Expression"
  }
  Parameter {
    Name                    "Frequency"
    Value                   [1.0]
    String                  "1"
    StringType              "Expression"
  }
  Parameter {
    Name                    "Phase"
    Value                   [0.0]
    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.
```

Table A-20: Model.rtw Block Specific Records (Continued)

```
  Name                    "sin_h"
  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"
}
```

Block Type: Step (with a step time of 2)

```
Parameter {
  Name                    "Time"
  Value                   [2.0]
  String                  "2"
  StringType              "Expression"
```

Table A-20: Model.rtw Block Specific Records (Continued)

```
}
Parameter {
  Name                    "Before"
  Value                   [0.0]
  String                  "0"
  StirngType              "Expression"
}
Parameter {
  Name                    "After"
  Value                   [1.0]
  String                  "1"
  StringType              "Expression"
}
```

Block Type: `StateSpace`

See Model.rtw Linear Block Specific Records on page A-75.

Block Type: `Sum`

```
ParamSettings {
  Inputs
}
```

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

Block Type: `SubSystem`

```
ParamSettings {
  SystemIdx
  StatesWhenEnabling
  TriggerBlock
  SystemContStates
}
```

`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).

Table A-20: Model.rtw Block Specific Records (Continued)

```
Parameter {
  Name                    "Threshold"
  Value                   [0.0]
  String                  "0"
  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
}
```

Block Type: `ToWorkspace`

```
ParamSettings {
  VariableName            Name of variable used to save scope data.
```

Table A-20: Model.rtw Block Specific Records (Continued)

```
    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 Linear Block Specific Records on page A-75.

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 {
    InitialInput            [0]
    BufferSize              [1024]
  }
  NumIWorkDefines           1
  IWorkDefine {
    Name                    "BufferIndices"
    Width                   4
  }
  NumPWorkDefines           1
  PWorkDefine {
    Name                    "TUbuffer"
    Width                   2
  }
```

Table A-20: Model.rtw Block Specific Records (Continued)

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`
`}`	

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                 [-1.0]
  String                "-1"
  StringType            "Expression"
}
Parameter {
  Name                  "MaxMinusMin"
  Value                 [2.0]
  String                ""
  StringType            "Computed"
}
NumIWorkDefines         1
IWorkDefine {
  Name                  "RandSeed"
  Width                 1
}
NumRWorkDefines         1
RWorkDefine {
```

Table A-20: Model.rtw Block Specific Records (Continued)

```
    Name                    "NextOutput"
    Width                   1
  }
```

Block Type: `UnitDelay` (example of a unit delay with an initial condition of 0).

```
  Parameter {
    Name                    "X0"
    Value                   [0.0]
    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                   [10.0]
    String                  "10"
    StringType              "Expression"
  }
  ParamSettings {
    InitialInput            [0.0]
    BufferSize              [1024.0]
  }
  NumIWorkDefines           1
  IWorkDefine {
    Name                    "BufferIndices"
    Width                   4
  }
  NumPWorkDefines           1
  PWorkDefine {
    Name                    "TUbuffer"
    Width                   2
```

Table A-20: Model.rtw Block Specific Records (Continued)

```
}
```

Block Type: `Width`

No block specific records.

Block Type: `ZeroPole`

See Model.rtw Linear Block Specific Records on page A-75.

Block Type: `ZeroOrderHold`

No block specific records.

Linear Block Specific Records

The following table describes the block specific records written for the Simulink linear blocks. These fields are common to all the discrete and continuous state space, transfer function, and discrete filter blocks given above.

Table A-21: Model.rtw Linear Block Specific Records

`Parameter {`	Vector of nonzero terms of the `A` matrix if realization is sparse, otherwise it is the first row of the `A` matrix.
`  Name`	`"Amatrix"`
`  Value`	Vector that could be of zero length.
`  String`	`""`
`  StringType`	`"Computed"`
`}`	
`Parameter {`	Vector of nonzero terms of the `B` matrix.
`  Name`	`"Bmatrix"`
`  Value`	Vector that could be of zero length.
`  String`	`""`
`  StringType`	`"Computed"`
`}`	
`Parameter {`	Vector of nonzero terms of the `C` matrix if realization is sparse, else it is the full `C` 2-D matrix.
`  Name`	`"Cmatrix"`
`  Value`	Vector that could be of zero length.
`  String`	`""`
`  StringType`	`"Computed"`
`}`	
`Parameter {`	Vector of nonzero terms of the `D` matrix.
`  Name`	`"Dmatrix"`
`  Value`	Vector that could be of zero length.
`  String`	`""`
`  StringType`	`"Computed"`
`}`	
`Parameter {`	Initial condition vector or `[]`.
`  Name`	`"X0"`

Table A-21: Model.rtw Linear Block Specific Records (Continued)

`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.
`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 that this directive causes the Target Language Compiler to terminate regardless of the `–mnumber` 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 that `%trace` directives are not considered errors and do not cause the Target Language Compiler to stop processing.

%warning directive: %s

The `%warning` directive produces this error message and displays the text following the `%warning` directive. Note that `%warning` directives are not considered errors and do not cause the Target Language Compiler to stop processing.

A %implements directive must appear within a block template file and must match the %language and type specified

A block template file was found, but it did not contain a `%implements` directive. A `%implements` directive is required to ensure that the correct language and type are implemented by this block template file. See "Object-Oriented Facility for Generating Target Code" on page 2-26 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"
       Value2
   }
}
block
{
   Name"Abc3"
   Parameter {
       Name"foo"
       Value3
   }
}

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 );
```

Bad directory (*dirname*) in -O: *filename*

The `-O` option was not a valid directory.

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

File name too long (directory = 'dirname', name = 'filename')

The specified filename was too long.

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.

Function values are not allowed

Attempt to use a TLC function as a variable.

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.

Indices may not be negative

An index used in a `[]` expression must be a nonnegative integer.

Invalid handle

An invalid handle was passed to the Target Language Compiler Server Mode

Invalid identifier range, the leading strings *string1* and *string2* must match

When using a range of signals, for example, `u1:u10`, the identifier in the first argument did not match the identifier in the second.

Invalid identifier range, the lower bound (%d) must be less than the upper bound (%d)

When using a range of signals, for example, `u1:u10`, the lower bound was higher than the upper bound.

Invalid type for unary *operator*

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.

License checkout failed. In order to generate Ada, you must purchase a license. Please contact your MathWorks sales representative

This error occurs when you do not have a valid license available for generating Ada code.

Macro expansion values are not allowed

It is illegal to use a macro as a TLC variable.

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 strings or cells of strings may be used as the argument to Query and ExecString

A cell containing non-string data was passed as the third argument to Query or ExecString in Server Mode.

Only vectors of the same length as the existing vector value can be assigned using the [] notation

When using the `[]` notation to replace a row of a matrix, the row must be a vector of the same length as the existing rows.

Output file *identifier* opened with %openfile was not closed

Output files opened with `%openfile` must be closed with `%closefile`. `identifier` is the name of the variable specified in the `%openfile` directive.

Ranges, identifier ranges, and repeat values cannot be repeated

You cannot repeat a range, idrange, or repeat value. This prevents things like `[1@2@3]`.

String cannot modify the setting for the command line switch '*-switch*'

The switch specified from the TLC file cannot be changed except from the command line.

String cannot modify the setting for the command line switch '*-switch*'

`%setcommandswitch` does not recognize the specified switch, or cannot modify it (e.g. `-r` cannot be modified).

'*String*' is not a recognized user defined property of this handle

Only double and character arrays can be converted from Matlab to the Target Language Compiler. This can occur if the Matlab function does not return a value (see `%%matlab`).

Syntax error

The indicated line contains a syntax error, See Chapter , "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
`%foreach`	`%endforeach`	Looping
`%roll`	`%endroll`	Loop Rolling
`%with`	`%endwith`	Scoping directive

`%switch`	`%endswitch`	Switch directive
`%function`	`%endfunction`	Function declaration directive

The error is reported on the line that opens the scope and has no matching `end` scope.

Note Nested scopes must be closed before their parent scopes. Failure to include an `end` for a nested scope often causes this error, as in:

```
%if Block.Name == "Sin 3"
    %foreach idx = Block.Width
%endif%% Error reported here that the %foreach was not terminated
```

The construct %%matlab *functionname*(...) construct is illegal in standalone tlc

You cannot call MATLAB from stand-alone TLC.

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 that 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 "%roll" on page 2-23 for a complete description of this command.

The first argument of a ? : operator must be a Boolean expression

The `? :` operator must have a Boolean expression as its first operand.

The first argument to GENERATE or GENERATE_TYPE must be a valid scope

When calling `GENERATE` or `GENERATE_TYPE`, the first argument must be a valid scope. See "GENERATE and GENERATE_TYPE Functions" on page 2-27 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 "%roll" on page 2-23 for a complete discussion of the `%roll` construct.

The roll argument to %roll must be a nonempty vector of numbers or ranges

When using `%roll`, the `roll` vector cannot be empty and must contain Numbers or Ranges of Numbers. See "%roll" on page 2-23 for a complete discussion of the `%roll` construct.

The second value in a Range must be greater than the first value

When using a range, for example, `1:10`, the lower bound was higher than the upper bound.

The specified index (*index*) was out of the range 0 to number-of-elements – 1

This error occurs when indexing into any nonscalar beyond the end of the variable. For example:

```
%assign x = [1 2 3]
%assign y = x[3]
```

This example would cause this error. Remember, in the Target Language Compiler, array indices start at 0 and go to the number of elements minus 1.

The STRINGOF built-in function expects a vector of numbers as its argument

The `STRINGOF` function expects a vector of numbers. The function treats each number as the ASCII value of a valid character.

The SYSNAME built-in function expects an input string of the form <xxx>/yyy

The `SYSNAME` function takes a single string of the form `<xxx>/yyy` as it appears in the `.rtw` file and returns a vector of two strings `xxx` and `yyy`. If the input argument does not match this format, it returns this error.

The threshold on a %roll statement must be a single number

When using `%roll`, the roll threshold specified must be a single number. See "%roll" on page 2-23 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 are no more free contexts. Use TLC('close', HANDLE) to freeup a context

The global context table has filled up.

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 "GENERATE and GENERATE_TYPE Functions" on

page 2-27 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 page 2-10.

Unrecognized command line switch passed to *string*: *switch*

When querying the current state of a switch, the switch specified was not recognized.

Unrecognized directive "*directive-name*" seen

An illegal `%` directive was encountered. The valid directives are:

`%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`
`%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.

Values of type Special, Macro Expansion, Function, File, Full Identifier, and Index cannot be converted to MATLAB variables

The specified type cannot be converted to MATLAB variables.

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

Block Target File Mapping

The following table lists Simulink block types and the corresponding target file that the Target Language Compiler uses to generate the corresponding code.

Block Type	Target Filename
Abs	`absval.tlc`
Backlash	`backlash.tlc`
Clock	`clock.tlc`
CombinatorialLogic	`cmblogic.tlc`
ComplexToMagnitudeAngle	`cmpx2maan.tlc`
ComplexToRealImag	`cmpx2reim.tlc`
Constant	`constant.tlc`
DataStoreMemory	`virtual.tlc`
DataStoreRead	`dsread.tlc`
DataStoreWrite	`dswrite.tlc`
DataTypeConversion	`dtypcnvb.tlc`
Demux	`virtual.tlc`
Derivative	`deriv.tlc`
DigitalClock	`digclock.tlc`
DiscreteFilter	`dfilter.tlc`
DiscreteIntegrator	`dintegrt.tlc`
DiscretePulseGenerator	`dpulsegen.tlc`
DiscreteStateSpace	`dss.tlc`
DiscreteTransferFcn	`dtf.tlc`
DiscreteZeroPole	`dzp.tlc`
Display	`display.tlc`

Block Type	Target Filename
ElementaryMath	`elmath.tlc`
EnablePort	`enabport.tlc`
FlatRoller	`flat_roller.tlc`
From	`virtual.tlc`
FromFile	`fromfile.tlc`
FromWorkspace	`fromwks.tlc`
Fcn	`fcn.tlc`
Gain	`gain.tlc`
Goto	`virtual.tlc`
GotoTagVisibility	`virtual.tlc`
Ground	`virtual.tlc`
HitCross	`hitcross.tlc`
InitialCondition	`initcond.tlc`
Inport	`inport.tlc`
Integrator	`integrat.tlc`
Logic	`logic.tlc`
Lookup	`look_up.tlc`
Lookup2D	`lookup2d.tlc`
MagnitudeAngleToComplex	`maan2cmpx`
MATLABFcn	`matlabfn.tlc`
Math	`bmathfcn.tlc`
Memory	`mem.tlc`
Merge	`merge.tlc`

Block Type	Target Filename
MinMax	`minmax.tlc`
MultiPortSwitch	`mpswitch.tlc`
Mux	`virtual.tlc`
Outport	`outport.tlc`
Probe	`probe.tlc`
Product	`product.tlc`
Quantizer	`quantize.tlc`
RandomNumber	`randnum.tlc`
RateLimiter	`ratelim.tlc`
RealImagToComplex	`reim2cmpx.tlc`
RelationalOperator	`relop.tlc`
Relay	`relay.tlc`
ResetIntegrator	`resetint.tlc`
Roller	`roller.tlc`
Rounding	`roundfcn.tlc`
Saturate	`saturate.tlc`
Scope	`scope.tlc`
Selector	`virtual.tlc`
SignalGenerator	`siggen.tlc`
Signum	`signum.tlc`
Sin	`sin_wave.tlc`
“S-function”	`gensfun.tlc`
StateSpace	`css.tlc`

Block Type	Target Filename
Step	`step.tlc`
Stop	`stop.tlc`
SubSystem	`subsystm.tlc`
Sum	`sum.tlc`
Switch	`swtchblk.tlc`
ToFile	`tofile.tlc`
ToWorkspace	`towks.tlc`
Terminator	`virtual.tlc`
TransferFcn	`ctf.tlc`
TransportDelay	`tdelay.tlc`
TriggerPort	`trigport.tlc`
Trigonometry	`trigfcn.tlc`
UniformRandomNumber	`urandnum.tlc`
UnitDelay	`delay.tlc`
VariableTransportDelay	`vtdelay.tlc`
Width	`width.tlc`
ZeroPole	`czp.tlc`
ZeroOrderHold	`zoh.tlc`

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

Z