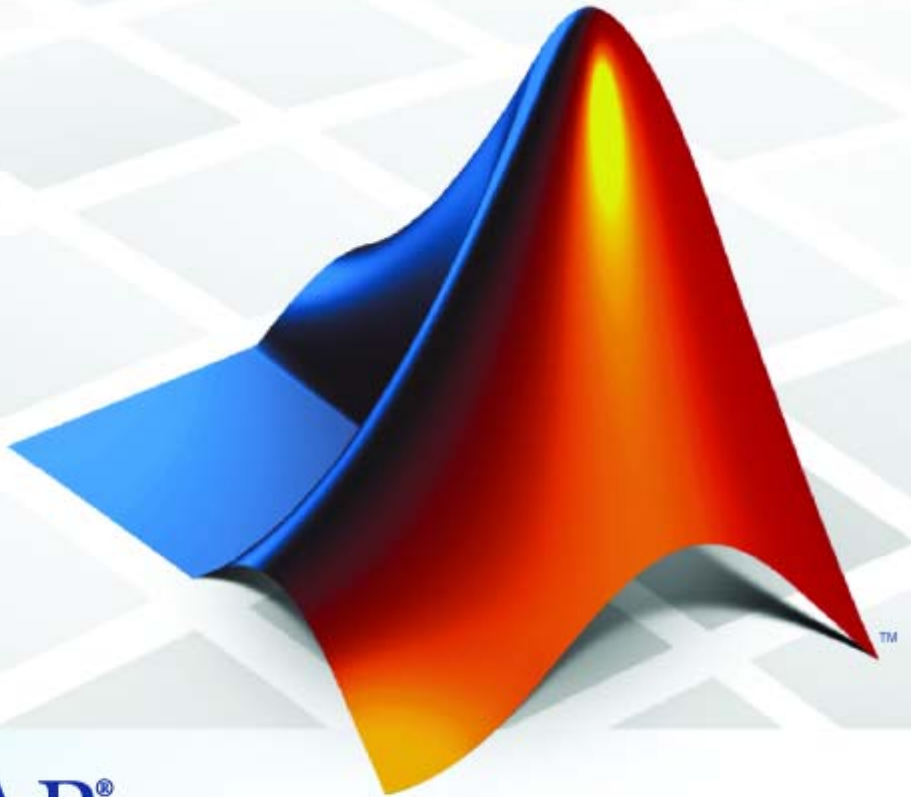


Simscape™ 3

Language Guide



MATLAB®
& **SIMULINK®**

How to Contact The MathWorks



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html Technical Support



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



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Simscape™ Language Guide

© COPYRIGHT 2008–2010 by The MathWorks, Inc.

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

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

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

October 2008	Online only	New for Version 3.0 (Release 2008b)
March 2009	Online only	Revised for Version 3.1 (Release 2009a)
September 2009	Online only	Revised for Version 3.2 (Release 2009b)
March 2010	Online only	Revised for Version 3.3 (Release 2010a)

About the Simscape Language

1

What Is the Simscape Language?	1-2
Typical Tasks	1-3
About Simscape Files	1-5
Simscape File Type	1-5
Model Types	1-5
Basic File Structure	1-8
Basic Simscape Grammar	1-10
Creating a New Physical Domain	1-12
When to Define a New Physical Domain	1-12
How to Define a New Physical Domain	1-13
Creating Custom Components	1-14
Component Types and Prerequisites	1-14
How to Create a New Behavioral Model	1-14
Defining Domain-Wide Parameters	1-15
Adding a Custom Block Library	1-16

Writing Simscape Files

2

Declaring Domains and Components	2-2
Declaration Grammar	2-2
Definitions	2-3
Member Declarations	2-4
Member Summary	2-5
Declaring a Member as a Value with Unit	2-5

Declaring Through and Across Variables for a Domain . . .	2-6
Declaring Component Variables	2-7
Declaring Component Parameters	2-8
Declaring Domain Parameters	2-10
Declaring Component Nodes	2-10
Declaring Component Inputs and Outputs	2-11
Example — Declaring a Mechanical Rotational Domain . .	2-12
Example — Declaring a Spring Component	2-13
Defining Component Setup	2-15
Setup Section Purpose	2-15
Validating Parameters	2-17
Computing Derived Parameters	2-17
Setting Initial Conditions	2-18
Defining Relationship Between Component Variables and Nodes	2-19
Defining Component Equations	2-20
Equation Section Purpose	2-20
Equation Grammar	2-21
Specifying Mathematical Equality	2-22
Use of Relational Operators in Equations	2-23
Equation Dimensionality	2-25
Equation Continuity	2-26
Using Conditional Expressions in Equations	2-27
Using Intermediate Terms in Equations	2-28
Examples of Equations	2-39
Putting It Together — Complete Component	
Examples	2-41
Mechanical Component Example — Spring	2-41
Electrical Component Example — Ideal Capacitor	2-42
No-Flow Component Example — Voltage Sensor	2-44
Grounding Component Example — Electrical Reference . .	2-45
Working with Domain Parameters	2-47
Propagation of Domain Parameters	2-47
Source Components	2-48
Propagating Components	2-48
Blocking Components	2-49
Example of Using Domain Parameters	2-49

Attribute Lists	2-56
Attribute Types	2-56
Model Attributes	2-56
Member Attributes	2-57
Subclassing and Inheritance	2-59

Using Simscape Files in Block Diagrams

3

Adding Custom Block Libraries Generated from Simscape Component Files	3-2
Workflow Overview	3-2
Organizing Your Simscape Files	3-3
Using Source Protection for Simscape Files	3-3
Converting Your Simscape Files	3-4
Customizing the Library Name and Appearance	3-5
Customizing the Library Icon	3-7
Example — Creating and Customizing Block Libraries ...	3-8
Specifics of Using Customized Domains	3-9
Customizing the Block Name and Appearance	3-11
Default Block Display	3-11
How to Customize the Block Name	3-13
How to Describe the Block Purpose	3-14
How to Specify Meaningful Names for the Block Parameters	3-15
How to Customize the Names and Locations of the Block Ports	3-16
How to Customize the Block Icon	3-18
Example — Customized Block Display	3-20
Checking File and Model Dependencies	3-22
Why Check File and Model Dependencies?	3-22
Checking Dependencies of Protected Files	3-23
Checking Simscape File Dependencies	3-23
Checking Library Dependencies	3-24
Checking Model Dependencies	3-24

Case Study — Creating a Basic Custom Block	
Library	3-26
Getting Started	3-26
Building the Custom Library	3-27
Adding a Block	3-27
Adding Detail to a Component	3-28
Adding a Component with an Internal Variable	3-30
Customizing the Block Icon	3-32
Case Study — Creating an Electrochemical Library ..	3-33
Getting Started	3-33
Building the Custom Library	3-34
Defining a New Domain	3-34
Structuring the Library	3-37
Defining a Reference Component	3-37
Defining an Ideal Source Component	3-38
Defining Measurement Components	3-39
Defining Basic Components	3-41
Defining a Cross-Domain Interfacing Component	3-44
Customizing the Appearance of the Library	3-46
Using the Custom Components to Build a Model	3-47
References	3-47

Index

About the Simscape Language

- “What Is the Simscape Language?” on page 1-2
- “Typical Tasks” on page 1-3
- “About Simscape Files” on page 1-5
- “Basic Simscape Grammar” on page 1-10
- “Creating a New Physical Domain” on page 1-12
- “Creating Custom Components” on page 1-14

What Is the Simscape Language?

The Simscape™ language extends the Simscape modeling environment by enabling you to create new components that do not exist in the Foundation library or in any of the add-on products. It is a dedicated textual language for modeling physical systems and has the following characteristics:

- Based on the MATLAB® programming language
- Contains additional constructs specific to physical modeling

The Simscape language makes modeling physical systems easier and more intuitive. It lets you define custom components as textual files, complete with parameterization, physical connections, and equations represented as acausal implicit differential algebraic equations (DAEs). The components you create can reuse the physical domain definitions provided with Simscape to ensure that your components are compatible with the standard Simscape components. You can also add your own physical domains. You can automatically build and manage block libraries of your Simscape components, enabling you to share these models across your organization.

Typical Tasks

Simscape block libraries contain a comprehensive selection of blocks that represent engineering components such as valves, resistors, springs, and so on. These prebuilt blocks, however, may not be sufficient to address your particular engineering needs. When you need to extend the existing block libraries, use the Simscape language to define customized components, or even new physical domains, as textual files. Then convert your textual components into libraries of additional Simscape blocks that you can use in your model diagrams. For more information on the modeling interface, see Chapter 3, “Using Simscape Files in Block Diagrams”.

The following table lists typical tasks along with links to background information and examples.

Task	Background Information	Examples
Create a custom component model based on equations	<p>“Creating Custom Components” on page 1-14</p> <p>“Declaring Domains and Components” on page 2-2</p> <p>“Defining Component Setup” on page 2-15</p> <p>“Defining Component Equations” on page 2-20</p>	<p>“Example — Declaring a Spring Component” on page 2-13</p> <p>“Mechanical Component Example — Spring” on page 2-41</p> <p>“Electrical Component Example — Ideal Capacitor” on page 2-42</p>
Add a custom block library to Simscape libraries	<p>“Adding Custom Block Libraries Generated from Simscape Component Files” on page 3-2</p> <p>“Using Source Protection for Simscape Files” on page 3-3</p> <p>“Customizing the Block Name and Appearance” on page 3-11</p>	<p>“Example — Creating and Customizing Block Libraries” on page 3-8</p> <p>“Example — Customized Block Display” on page 3-20</p>

Task	Background Information	Examples
Define a new domain, with associated Through and Across variables, and then use it in custom components	“Creating a New Physical Domain” on page 1-12 “Declaring Domains and Components” on page 2-2	“Example — Declaring a Mechanical Rotational Domain” on page 2-12 “Propagation of Domain Parameters” on page 2-47
Create a component that supplies domain-wide parameters (such as fluid temperature) to the rest of the model	“Working with Domain Parameters” on page 2-47	“Source Components” on page 2-48

About Simscape Files

In this section...
“Simscape File Type” on page 1-5
“Model Types” on page 1-5
“Basic File Structure” on page 1-8

Simscape File Type

The Simscape file is a dedicated file type in the MATLAB environment. It has the extension `.ssc`.

The Simscape file contains language constructs that do not exist in MATLAB. They are specific to modeling physical objects, as described in Chapter 2, “Writing Simscape Files”. However, the Simscape file incorporates the basic MATLAB programming syntax at the lowest level.

Simscape files must reside in a `+package` directory on the MATLAB path:

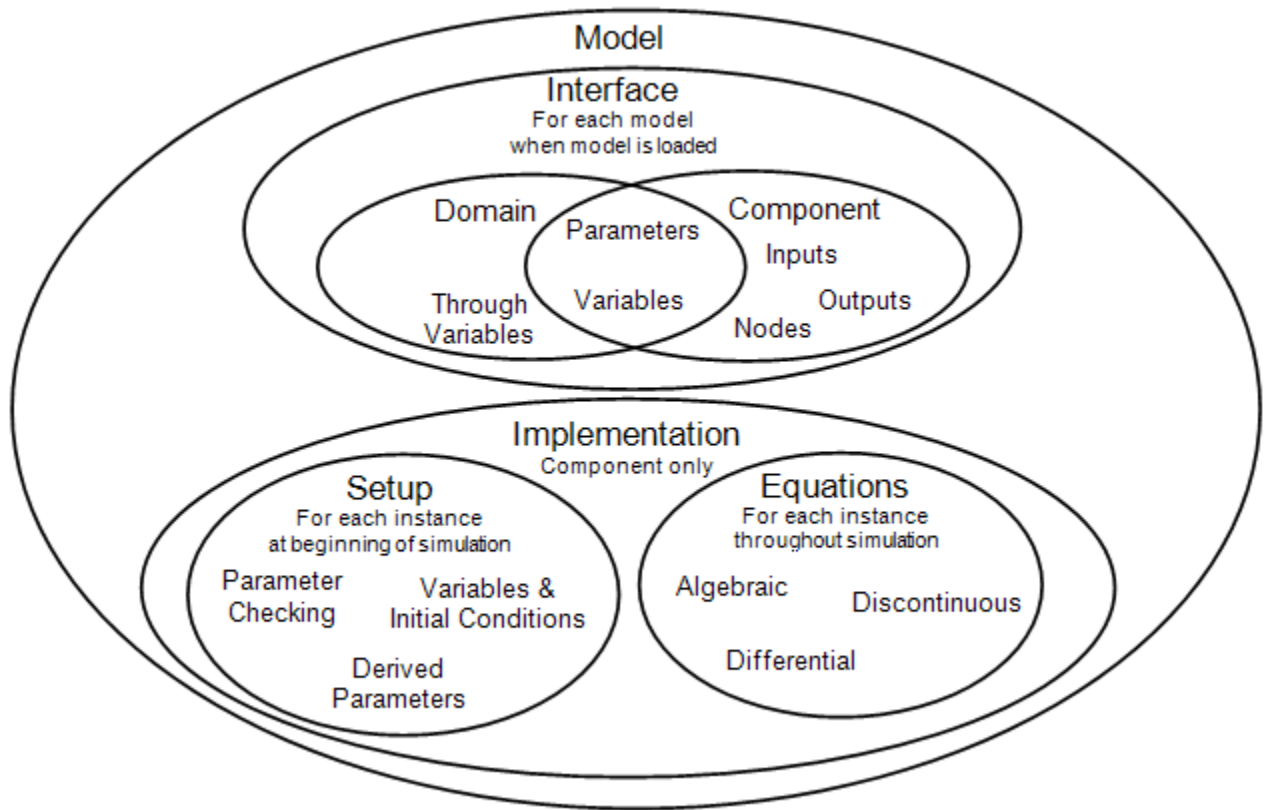
- `directory_on_the_path/+MyPackage/MyComponent.ssc`
- `directory_on_the_path/+MyPackage/+Subpackage/.../MyComponent.ssc`

For more information on packaging your Simscape files, see “Organizing Your Simscape Files” on page 3-3.

Model Types

There are two types of Simscape files, corresponding to the two model types:

- *Domain* models describe the physical domains through which components models exchange energy and data. These physical domains correspond to port types, for example, translational, rotational, hydraulic, and so on.
- *Component* models describe the physical components that you want to model, that is, they correspond to Simscape blocks.



A Simscape file splits the model description into the following pieces:

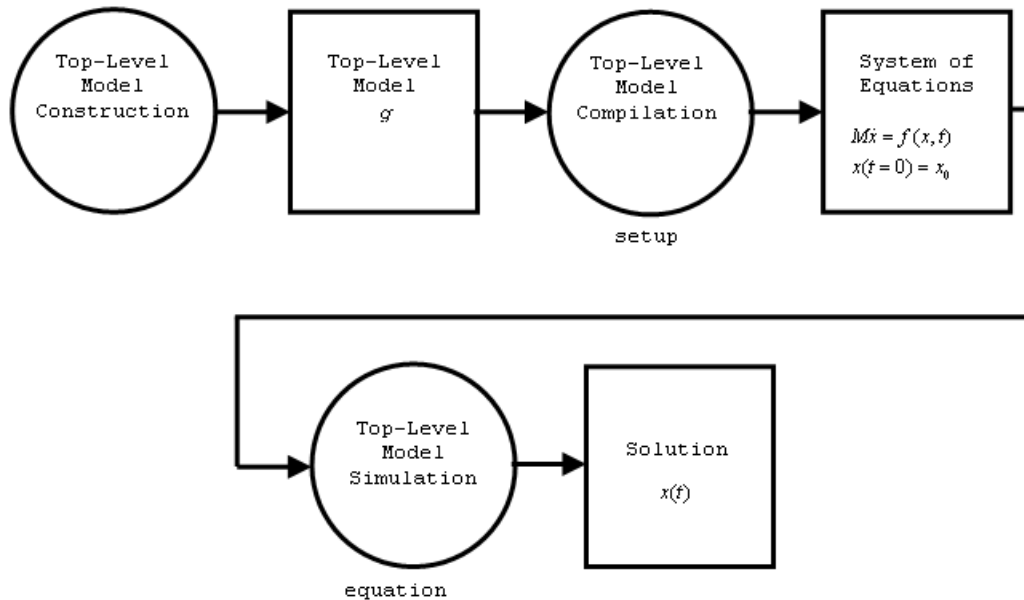
- *Interface or Declaration* — Declarative section similar to the MATLAB class system declarations:
 - For domain models, declares variables (Across and Through) and parameters
 - For component models, declares nodes, inputs and outputs, parameters, and variables
- *Implementation* (only for component models) — Describes run-time functionality of the model. Implementation consists of two sections:

- *Setup* — Performs initialization and setup. Executed once for each instance of the component in the top-level model during model compilation.
- *Equation* — Describes underlying equations. Executed throughout simulation.

Like the MATLAB class system, these constructs and functions act on a specific instance of the class.

Unlike the MATLAB class system:

- The object is not passed as the first argument to function. This reduces syntax with no loss of functionality.
- These functions have specific roles in the component lifecycle, as shown in the following diagram.



Component Instance Lifecycle

Phase	Steps
Top-Level Model Construction	<ol style="list-style-type: none">1 Invokes file name from MATLAB to construct component instance2 Adds component instance to top-level model3 Sets parameters on component instance4 Connects component instance to other members of the top-level model
Top-Level Model Compilation	<ol style="list-style-type: none">5 Calls the <code>setup</code> function once for each component instance in the top-level model
Top-Level Model Simulation	<ol style="list-style-type: none">6 (Conceptually) calls the <code>equations</code> function for each component instance in the top-level model repeatedly throughout the simulation

Basic File Structure

Each model is defined in its own file of the same name with a `.ssc` extension. For example, `MyComponent` is defined in `MyComponent.ssc`. A model may be a domain model or a component model. Each Simscape file starts with a line specifying the model class and identifier:

```
ModelClass Identifier
```

where

- *ModelClass* is either domain or component
- *Identifier* is the name of the model

For example:

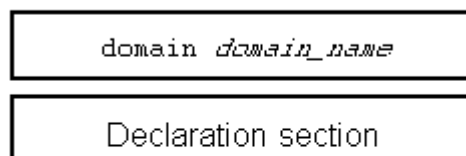
```
domain rotational
```

or

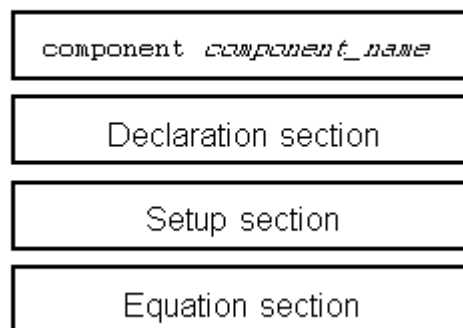
```
component spring
```

The basic file structure for domain models and component models is similar.

Domain file structure



Component file structure



Basic Simscape Grammar

The following table describes the Simscape file grammar. Curly braces {} indicate optional elements. The pipe | indicates a logical OR. Required characters and keywords are [blue](#).

Model	ModelClass { (AttributeList) Space } Identifier { < Identifier } Separator DeclarationSection Separator { SetupSection Separator } { EquationSection Separator } end
ModelClass	domain component
DeclarationSection	DeclarationBlock { Separator DeclarationBlock }
DeclarationBlock	MemberClass { (AttributeList) } Separator Identifier = Expression Separator { Identifier = Expression Separator } end
MemberClass	variables parameters inputs outputs nodes
SetupSection	function setup SetupStatement Separator { SetupStatement Separator } end
EquationSection	equations ExpressionList end
ExpressionList	EquationExpression { Separator EquationExpression }
EquationExpression	Expression CondExpression LetExpression
CondExpression	if Expression ExpressionList { elseif Expression ExpressionList } else ExpressionList end

LetExpression	<code>let</code> DeclarationList <code>in</code> ExpressionList <code>end</code>
DeclarationList	Declaration { Separator Declaration }
Declaration	LetValue = EquationExpression
LetValue	Identifier IdentifierList

where

AttributeList	Attribute list as defined in the MATLAB class system grammar
Expression	MATLAB expression
Identifier	MATLAB identifier
IdentifierList	MATLAB identifier list in square brackets []
Separator	Comma, semicolon, or newline
SetupStatement	A combination of assignment statements, simple if statements, and error functions
Space	Required white space, but not newline

Creating a New Physical Domain

In this section...
“When to Define a New Physical Domain” on page 1-12
“How to Define a New Physical Domain” on page 1-13

When to Define a New Physical Domain

A physical domain provides an environment, defined primarily by its Across and Through variables, for connecting the components in a Physical Network. Component nodes are typed by domain, that is, each component node is associated with a unique type of domain and can be connected only to nodes associated with the same domain.

You do not need to define a new physical domain to create custom components. Simscape software comes with several predefined domains, such as mechanical translational, mechanical rotational, electrical, hydraulic, and so on. These domains are included in the Foundation library, and are the basis of Simscape Foundation blocks, as well as those in Simscape add-on products (for example, SimHydraulics® or SimElectronics® blocks). If you want to create a custom component to be connected to the standard Simscape blocks, use the Foundation domain definitions. For a complete listing of the Foundation domains, see “Simscape Foundation Domains” in the *Simscape Reference*.

You need to define a new domain only if the Foundation domain definitions do not satisfy your modeling requirements. For example, to enable modeling electrochemical systems, you need to create a new domain with the appropriate Across and Through variables. If you need to model a thermohydraulic system, you can create a custom hydraulic domain that accounts for fluid temperature by supplying a domain-wide parameter (for an example, see “Propagation of Domain Parameters” on page 2-47).

Once you define a custom physical domain, you can use it for defining nodes in your custom components. These nodes, however, can be connected only to other nodes of the same domain type. For example, if you define a custom hydraulic domain as described above and then use it when creating custom components, you will not be able to connect these nodes with the regular

hydraulic ports of the standard Simscape blocks, which use the Foundation hydraulic domain definition.

How to Define a New Physical Domain

To define a new physical domain, you must declare the Through and Across variables associated with it. For more information, see “Basic Principles of Modeling Physical Networks” in the *Simscape User’s Guide*.

A domain file must begin with the `domain` keyword, followed by the domain name, and be terminated by the `end` keyword.

Domain files contain only the declaration section. Two declaration blocks are required:

- The Across variables declaration block, which begins with the `variables` keyword and is terminated by the `end` keyword. It contains declarations for all the Across variables associated with the domain. A domain model class definition can contain multiple Across variables, combined in a single `variables` block.
- The Through variables declaration block, which begins with the `variables(Balancing = true)` keyword and is terminated by the `end` keyword. It contains declarations for all the Through variables associated with the domain. A domain model class definition can contain multiple Through variables, combined in a single `variables(Balancing = true)` block.

For more information on declaring the Through and Across variables, see “Declaring Through and Across Variables for a Domain” on page 2-6.

The parameters declaration block is optional. If present, it must begin with the `parameters` keyword and be terminated by the `end` keyword. This block contains declarations for domain parameters. These parameters are associated with the domain and can be propagated through the network to all components connected to the domain. For more information, see “Working with Domain Parameters” on page 2-47.

For an example of a domain file, see “Example — Declaring a Mechanical Rotational Domain” on page 2-12.

Creating Custom Components

In this section...

- “Component Types and Prerequisites” on page 1-14
- “How to Create a New Behavioral Model” on page 1-14
- “Defining Domain-Wide Parameters” on page 1-15
- “Adding a Custom Block Library” on page 1-16

Component Types and Prerequisites

In physical modeling, there are two types of models:

- Behavioral — A model that is implemented based on its physical behavior, described by a system of mathematical equations. An example of a behavioral block implementation is the Variable Orifice block.
- Structural — A model that is constructed out of other blocks, connected in a certain way. An example of a structural block implementation is the 4-Way Directional Valve block (available with SimHydraulics block libraries), which is constructed based on four Variable Orifice blocks.

Simscape language lets you create new behavioral models when your design requirements are not satisfied by the libraries of standard blocks provided with Simscape and its add-on products.

You can then use these custom behavioral models either as standalone blocks or as building blocks for structural models. To create a new structural model, use masked subsystems. For more information, see “Creating Subsystems” in the *Simulink® User’s Guide*.

A prerequisite to creating components is having the appropriate domains for the component nodes. You can use Simscape Foundation domains or create your own, as described in “Creating a New Physical Domain” on page 1-12.

How to Create a New Behavioral Model

To create a new behavioral model, define a component model class by writing a component file.

A component file must begin with the `component` keyword, followed by the component name, and be terminated by the end keyword.

Component files typically contain three sections:

- **Declaration** — Contains all the member class declarations for the component, such as parameters, variables, nodes, inputs, and outputs. Each member class declaration is a separate declaration block, which begins with the appropriate keyword (corresponding to the member class) and is terminated by the end keyword. For more information, see the component-related sections in “Declaring Domains and Components” on page 2-2.
- **Setup** — Prepares the component for simulation. The body of the `setup` function can contain assignment statements, `if` and `error` statements, and `across` and `through` functions. The `setup` function is executed once for each component instance during model compilation. It takes no arguments and returns no arguments. For more information, see “Defining Component Setup” on page 2-15.
- **Equation** — Declares the component equations. These equations may be conditional, and are applied throughout the simulation. For more information, see “Defining Component Equations” on page 2-20.

Defining Domain-Wide Parameters

Another type of a custom block is an environment block that acts as a source of domain-wide parameters. For example, you can create a Hydraulic Temperature block that supplies the temperature parameter to the rest of the model.

Note The Foundation hydraulic domain does not contain a temperature parameter. You would have to create a customized hydraulic domain where this parameter is declared. Components using your own customized hydraulic domain cannot be connected with the components using the Simscape Foundation hydraulic domain. Use your own customized domain definitions to build complete libraries of components to be connected to each other.

You create environment components similar to behavioral components, by writing a component file that consists of the declaration, setup, and equation sections. However, to indicate that this component supplies the parameter value to the rest of the model, set the `Propagation` attribute of this component to `source`. For more information, see “Working with Domain Parameters” on page 2-47 and “Attribute Lists” on page 2-56.

Adding a Custom Block Library

Adding a custom block library involves creating new components (behavioral or environment). It may involve creating a new physical domain if the Simscape Foundation domain definitions do not satisfy your modeling requirements.

After you have created the textual component files, convert them into a library of blocks using the procedure described in “Adding Custom Block Libraries Generated from Simscape Component Files” on page 3-2. You can control the block names and appearance by using optional comments in the component file. For more information, see “Customizing the Block Name and Appearance” on page 3-11.

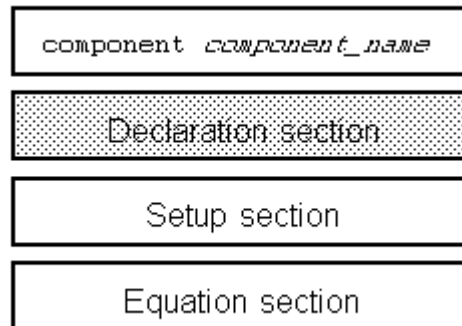
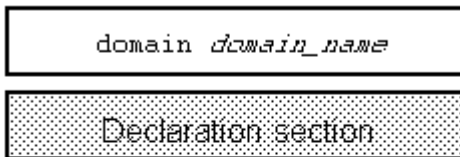
Writing Simscape Files

- “Declaring Domains and Components” on page 2-2
- “Defining Component Setup” on page 2-15
- “Defining Component Equations” on page 2-20
- “Putting It Together — Complete Component Examples” on page 2-41
- “Working with Domain Parameters” on page 2-47
- “Attribute Lists” on page 2-56
- “Subclassing and Inheritance” on page 2-59

Declaring Domains and Components

In this section...
“Declaration Grammar” on page 2-2
“Definitions” on page 2-3
“Member Declarations” on page 2-4
“Member Summary” on page 2-5
“Declaring a Member as a Value with Unit” on page 2-5
“Declaring Through and Across Variables for a Domain” on page 2-6
“Declaring Component Variables” on page 2-7
“Declaring Component Parameters” on page 2-8
“Declaring Domain Parameters” on page 2-10
“Declaring Component Nodes” on page 2-10
“Declaring Component Inputs and Outputs” on page 2-11
“Example — Declaring a Mechanical Rotational Domain” on page 2-12
“Example — Declaring a Spring Component” on page 2-13

Declaration Grammar



The following table describes the declaration section grammar. Curly braces {} indicate optional elements. The pipe | indicates a logical OR. Required characters and keywords are blue.

Declaration	DeclarationBlock { Separator DeclarationBlock }
DeclarationBlock	MemberClass { (AttributeList) } Separator Identifier = Expression Separator { Identifier = Expression Separator } end
MemberClass	variables parameters inputs outputs nodes

where

AttributeList	Attribute list as defined in the MATLAB class system grammar
Expression	MATLAB expression
Identifier	MATLAB identifier
Separator	Comma, semicolon, or newline

Definitions

The declaration section of a Simscape file may contain one or more member declarations.

Term	Definition
Member	<ul style="list-style-type: none"> • A member is a piece of a model's declaration. The collection of all members of a model is its declaration. • It has an associated data type and identifier. • Each member is associated with a unique <i>member class</i>. Additionally, members may have some specific attributes.
Member class	<ul style="list-style-type: none"> • A member class is the broader classification of a member. • The following is the set of member classes: variables (domain or component variables), parameters, inputs, outputs, nodes. • Two members may have the same type, but be of different member classes. For example, a parameter and an input may

Term	Definition
	have the same data type, but because they are of different member classes, they behave differently.

Member Declarations

The following rules apply to declaring members:

- Like the MATLAB class system, declared members appear in a declaration block:

```

<ModelClass> <Identifier>
    <MemberClass>
        % members here
    end
...
end
    
```

- Unlike the MATLAB class system, <MemberClass> may take on any of the available member classes and dictates the member class of the members defined within the block.
- Like the MATLAB class system, each declared member is associated with a MATLAB identifier, <Identifier>. Unlike the MATLAB class system, members *must* be declared with a right-hand side value.

```

<ModelClass> <Identifier>
    <MemberClass>
        <Identifier> = <Expression>;
        % more members
    end
...
end
    
```

- <Expression> on the right-hand side of the equal sign (=) is a MATLAB expression. It could be a constant expression, or a call to a MATLAB function.
- The MATLAB class of the expression is restricted by the class of the member being declared. Also, the data type of the expression dictates data type of the declared member.

Member Summary

The following table provides the summary of member classes.

Member Class	Applicable Model Classes	MATLAB Class of Expression	Expression Meaning	Writable
parameters	domain component	Numerical value with unit	Default value	Yes
variables	domain component	Double value with unit	Nominal value and default initial condition	Yes
inputs	component	Scalar double value with unit	Default value	No
outputs	component	Scalar double value with unit	Default value	No
nodes	component	Instance of a node associated with a domain	Type of domain	No

Note When a member is writable, it means that it can be assigned to in the `setup` function. Nodes are themselves not writable, but their writable members (parameters and variables) are.

Declaring a Member as a Value with Unit

In Simscape language, declaration members such as parameters, variables, inputs, and outputs, are represented as a value with associated unit. The syntax for a value with unit is essentially that of a two-member value-unit cell array:

```
{ value , 'unit' }
```

where `value` is a real matrix, including a scalar, and `unit` is a valid unit string, defined in the unit registry, or 1 (unitless). Depending on the member type, certain restrictions may apply. See respective reference pages for details.

For example, this is how you declare a parameter as a value with unit:

```
par1 = { value , 'unit' };
```

As in MATLAB, the comma is not required, and this syntax is equivalent:

```
par1 = { value 'unit' };
```

To declare a unitless parameter, you can either use the same syntax:

```
par1 = { value , '1' };
```

or omit the unit and use this syntax:

```
par1 = value;
```

Internally, however, this parameter will be treated as a two-member value-unit cell array { value , '1' }.

Declaring Through and Across Variables for a Domain

In a domain file, you have to declare the Through and Across variables associated with the domain. These variables characterize the energy flow and usually come in pairs, one Through and one Across. Simscape language does not require that you have the same number of Through and Across variables in a domain definition, but it is highly recommended. For more information, see “Basic Principles of Modeling Physical Networks” in the *Simscape User’s Guide*.

`variables` begins an Across variables declaration block, which is terminated by an `end` key word. This block contains declarations for all the Across variables associated with the domain. A domain model class definition can contain multiple Across variables, combined in a single `variables` block. This block is required.

Through variables are semantically distinct in that their values have to balance at a node: for each Through variable, the sum of all its values flowing into a branch point equals the sum of all its values flowing out. Therefore, a domain file must contain a separate declaration block for its Through variables, with the `Balancing` attribute set to `true`,

`variables(Balancing = true)` begins a Through variables definition block, which is terminated by an `end` key word. This block contains declarations for all the Through variables associated with the domain. A domain model class definition can contain multiple Through variables, combined in a single `variables(Balancing = true)` block. This block is required.

Each variable is defined as a value with unit:

```
domain_var1 = { value , 'unit' };
```

`value` is the initial value. `unit` is a valid unit string, defined in the unit registry. See “Example — Declaring a Mechanical Rotational Domain” on page 2-12 for more information.

Declaring Component Variables

When you declare Through and Across variables in a component, you are essentially creating instances of domain Through and Across variables. You declare a component variable as a value with unit by specifying an initial value and units commensurate with units of the domain variable.

The following example initializes the Through variable `t` (torque) as $0 \text{ N}\cdot\text{m}$:

```
variables
    t = { 0, 'N*m' };
end
```

Note After you declare component variables, you have to use `through` and `across` functions in the setup section to specify their relationship with component nodes.

You can also declare an internal component variable as a value with unit. You can use such internal variables in the setup and equation sections. Unlike component parameters, internal component variables do not appear in a block dialog box of the Simscape block generated from the component file.

Declaring Component Parameters

Component parameters let you specify adjustable parameters for the Simscape block generated from the component file. Parameters will appear in the block dialog box and can be modified when building and simulating a model.

You declare each parameter as a value with unit. Specifying an optional comment lets you control the parameter name in the block dialog box. For more information, see “How to Specify Meaningful Names for the Block Parameters” on page 3-15.

The following example declares parameter *k*, with a default value of 10 N*m/rad, specifying the spring rate of a rotational spring. In the block dialog box, this parameter will be named **Spring rate**.

```
parameters
    k = { 10, 'N*m/rad' };    % Spring rate
end
```

Specifying Parameter Units

When you declare a component parameter, use the units that make sense in the context of the block application. For example, if you model a solenoid, it is more convenient for the block user to input stroke in millimeters rather than in meters. When a parameter is used in the setup and equation sections, Simscape unit manager handles the conversions.

With temperature units, however, there is an additional issue of whether to apply linear or affine conversion (see “Thermal Unit Conversions” in the *Simscape User’s Guide*). Therefore, when you declare a parameter with temperature units, you can specify only nonaffine units (kelvin or rankine). When the block user enters the parameter value in affine units (Celsius or Fahrenheit), this value is automatically converted to the units specified in the parameter declaration. By default, affine conversion is applied. If a parameter specifies relative, rather than absolute, temperature (in other words, a change in temperature), set its `Conversion` attribute to `relative` (for details, see “Member Attributes” on page 2-57).

Note Member attributes apply to a whole DeclarationBlock. If some of your parameters are relative and others are absolute, declare them in separate blocks. You can have more than one declaration block of the same member type within a Simscape file.

Case Sensitivity

Simscape language is case-sensitive. This means that member names may differ only by case. However, Simulink software is not case-sensitive. Simulink parameter names (that is, parameter names in a block dialog box) must be unique irrespective of case. Therefore, if you declare two parameters whose names differ only by case, such as

```
component MyComponent
  parameters
    A = 0;
    a = 0;
  end
end
```

you will not be able to generate a block from this component.

However, if one of the parameters is private or hidden, that is, does not appear in the block dialog box,

```
component MyComponent
  parameters(Access=private)
    A = 0;
  end
  parameters
    a = 0;
  end
end
```

or if one is declared as a parameter and another as a variable, such as

```
component MyComponent
  variables
    A = 0;
```

```
end
parameters
    a = 0;
end
end
```

then there is no conflict in the Simulink namespace and no problem generating the block from the component source.

The case-sensitivity restriction applies only to component parameters, because other member types do not have an associated Simulink parameter, and are therefore completely case-sensitive.

Declaring Domain Parameters

Similar to a component parameter, you declare each domain parameter as a value with unit. However, unlike component parameters, the main purpose of domain parameters is to propagate the same parameter value to all or some of the components connected to the domain. For more information, see “Working with Domain Parameters” on page 2-47.

Declaring Component Nodes

Component nodes define the conserving ports of a Simscape block generated from the component file. The type of the conserving port (electrical, mechanical rotational, and so on) is determined by the type of its parent domain. The domain defines which Through and Across variables the port can transfer. Conserving ports of Simscape blocks can be connected only to ports associated with the same domain. For more information, see “Basic Principles of Modeling Physical Networks” in the *Simscape User’s Guide*.

When declaring nodes in a component, you have to associate them with an existing domain. You need to refer to the domain name using the full path starting with the top package directory. For more information on packaging your Simscape files, see “Adding Custom Block Libraries Generated from Simscape Component Files” on page 3-2.

The following example uses the syntax for the Simscape Foundation mechanical rotational domain:


```

nodes
    r = foundation.mechanical.rotational.rotational;
end

```

The name of the top-level package directory is `+foundation`. It contains a subpackage `+mechanical`, with a subpackage `+rotational`, which in turn contains the domain file `rotational.ssc`.

If you want to use your own customized rotational domain called `rotational.ssc` and located at the top level of your custom package directory `+MechanicalElements`, the syntax would be:

```

nodes
    r = MechanicalElements.rotational;
end

```

Note Components using your own customized rotational domain cannot be connected with the components using the Simscape Foundation mechanical rotational domain. Use your own customized domain definitions to build complete libraries of components to be connected to each other.

Specifying an optional comment lets you control the port label and location in the block icon. For more information, see “How to Customize the Names and Locations of the Block Ports” on page 3-16. In the following example, the electrical conserving port will be labelled `+` and will be located on the top side of the block icon.

```

nodes
    p = foundation.electrical.electrical; % +:top
end

```

Declaring Component Inputs and Outputs

In addition to conserving ports, Simscape blocks can contain Physical Signal input and output ports, directional ports that carry signals with associated units. These ports are defined in the `inputs` and `outputs` declaration blocks of a component file. Each input or output is defined as a value with unit.

Specifying an optional comment lets you control the port label and location in the block icon. For more information, see “How to Customize the Names and Locations of the Block Ports” on page 3-16.

The following example declares an input port `s`, with a default value of 1 Pa, specifying the control port of a hydraulic pressure source. In the block diagram, this port will be named **Pressure** and will be located on the top side of the block icon.

```
inputs
    s = { 1, 'Pa' }; % Pressure:top
end
```

Example – Declaring a Mechanical Rotational Domain

The following file, named `rotational.ssc`, declares a mechanical rotational domain, with angular velocity as an Across variable and torque as a Through variable.

```
domain rotational
% Define the mechanical rotational domain
% in terms of across and through variables

variables
    w = { 1, 'rad/s' }; % angular velocity
end

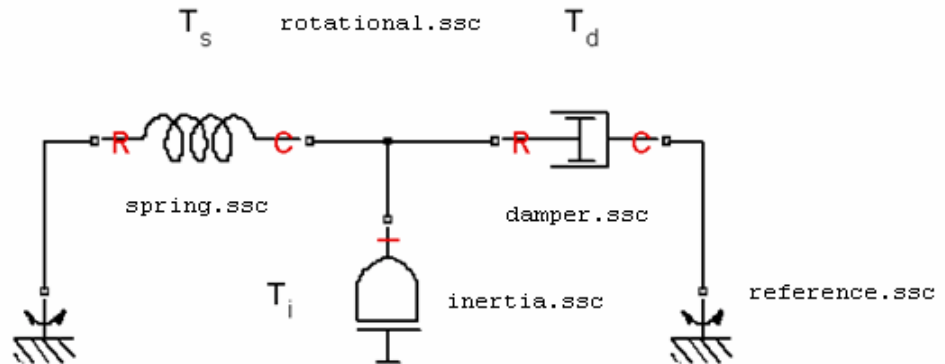
variables(Balancing = true)
    t = { 1, 'N*m' }; % torque
end

end
```

Note This domain declaration corresponds to the Simscape Foundation mechanical rotational domain. For a complete listing of the Foundation domains, see “Simscape Foundation Domains” in the *Simscape Reference*.

Example – Declaring a Spring Component

The following diagram shows a network representation of a mass-spring-damper system, consisting of four components (mass, spring, damper, and reference) in a mechanical rotational domain.



The domain is declared in a file named `rotational.ssc` (see “Example — Declaring a Mechanical Rotational Domain” on page 2-12). The following file, named `spring.ssc`, declares a component called `spring`. The component contains:

- Two rotational nodes, `r` and `c` (for rod and case, respectively)
- Parameter `k`, with a default value of $10 \text{ N}\cdot\text{m}/\text{rad}$, specifying the spring rate
- Through and Across variables, torque `t` and angular velocity `w`, later to be related to the Through and Across variables of the rotational domain
- Internal variable `theta`, with a default value of 0 rad , specifying relative angle, that is, deformation of the spring

```
component spring
  nodes
    r = foundation.mechanical.rotational.rotational;
    c = foundation.mechanical.rotational.rotational;
  end
  parameters
```

```
        k = { 10, 'N*m/rad' }; % spring rate
    end
    variables
        theta = { 0, 'rad' }; % introduce new variable for spring deformation
        t = { 0, 'N*m' }; % torque through
        w = { 0, 'rad/s' }; % velocity across
    end
    % setup here
    % equations here
end
```

Note This example shows only the declaration section of the spring component. For a complete file listing of a spring component, including the setup and equations, see “Mechanical Component Example — Spring” on page 2-41.

Defining Component Setup

In this section...

“Setup Section Purpose” on page 2-15

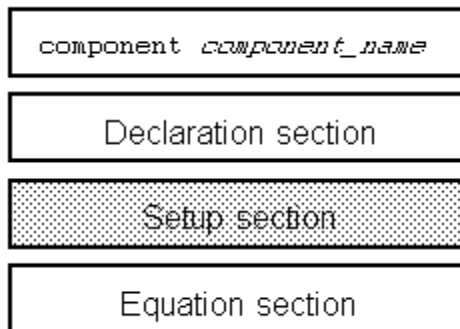
“Validating Parameters” on page 2-17

“Computing Derived Parameters” on page 2-17

“Setting Initial Conditions” on page 2-18

“Defining Relationship Between Component Variables and Nodes” on page 2-19

Setup Section Purpose



The setup section of a Simscape file follows the declaration section and consists of the function named `setup`. The `setup` function is executed once for each component instance during model compilation. It takes no arguments and returns no arguments.

Note Setup is not a constructor; it prepares the component for simulation.

Use the `setup` function for the following purposes:

- Validating parameters

- Computing derived parameters
- Setting initial conditions
- Relating variables and nodes to one another by using `across` and `through` functions

The following rules apply:

- The `setup` function is executed as regular MATLAB code.
- All members declared in the component are available by their name, for example:

```
component MyComponent
    parameters
        p = {1, 'm' };
    end
    [...]
    function setup
        disp( p ); % during compilation, prints value of p
                  % for each instance of MyComponent in the model
    [...]
end
```

- All members (such as variables, parameters) that are externally writable are writable within `setup`. See “Member Summary” on page 2-5 for more information.
- Local MATLAB variables may be introduced in the `setup` function. They are scoped only to the `setup` function.

The following restrictions apply:

- Command syntax is not supported in the `setup` function. You must use the function syntax. For more information, see “Command vs. Function Syntax” in the *MATLAB Programming Fundamentals* documentation.
- Persistent and global variables are not supported. For more information, see “Types of Variables” in the *MATLAB Programming Fundamentals* documentation.
- MATLAB system commands using the `!` operator are not supported.

- try-end and try-catch-end constructs are not supported.
- Passing declaration members to external MATLAB functions, for example, `my_function(param1)`, is not supported. You can, however, pass member values to external functions, for example, `my_function(param1.value('unit'))`.

Validating Parameters

The setup function validates parameters using simple if statements and the error function. For example:

```
component MyComponent
  parameters
    LowerThreshold = {1, 'm' };
    UpperThreshold = {1, 'm' };
  end
  [...]
  function setup
    if LowerThreshold > UpperThreshold
      error( 'LowerThreshold is greater than UpperThreshold' );
    end
  end
  [...]
end
```

Computing Derived Parameters

The setup function can override parameters by assigning to them. For example, it can verify that a parameter is not greater than the maximum allowed value, and if it is, issue a warning and assign the maximum allowed value to the parameter:

```
component MyComponent
  parameters
    MyParam = {1, 'm' };
  end
  [...]
  function setup
    MaxValue = {1, 'm' };
    if MyParam > MaxValue
      warning( 'MyParam is greater than MaxValue, overriding with MaxValue' );
    end
  end
end
```

```
        MyParam = MaxValue;
    end
end
[...]
```

Note Members are strongly typed. In the example above, `MaxValue` must have the same data type and compatible unit as `MyParam`. Otherwise, you will get an error.

Setting Initial Conditions

As you declare variables, values that you assign to them are their initial conditions. However, you can use the `setup` function to override these initial conditions by assigning the variable a new value, for example:

```
component MyComponent
    variables
        MyVariable = {1, 'm' };
    end
    [...]
    function setup
        MaxValue = {1, 'm' };
        if MyVariable > MaxValue
            warning( 'Initial value of MyVariable is greater than MaxValue, overriding with MaxValue' );
            MyVariable = MaxValue;    % MyVariable(t = 0) = MaxValue
        end
    end
end
[...]
```

Note Variables are also strongly typed. In the example above, `MaxValue` must have the same data type and compatible unit as `MyVariable`. Otherwise, you will get an error.

Defining Relationship Between Component Variables and Nodes

Use the `across` and `through` functions to establish relationship between the component variables and nodes. The `across` function is not strictly necessary because the same relationship for the `Across` variables could be established in the equation section, but it acts as shorthand and adds notation that clearly illustrates the relationship among the variables. The `through` function is the only way to establish relationship between the `Through` variables. These functions are especially helpful when the component has multiple nodes because they clearly indicate branches.

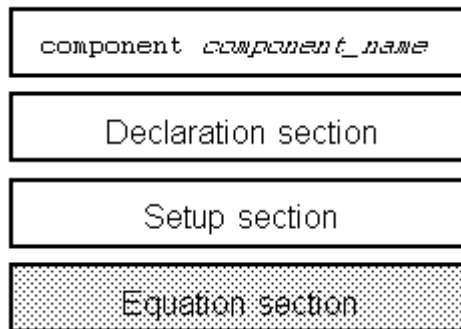
In the following example, `r` and `c` are rotational nodes, while `t` and `w` are component variables for torque and angular velocity, respectively. The setup section defines the relationship between the variables and nodes:

```
component spring
  nodes
    r = foundation.mechanical.rotational.rotational;
    c = foundation.mechanical.rotational.rotational;
  end
  [...]
  variables
    [...]
    t = { 0, 'N*m' };      % torque through
    w = { 0, 'rad/s' };   % velocity across
  end
  function setup
    through( t, r.t, c.t ); % t a through variable from r to c
    across(w, r.w, c.w );  % w an across variable from r to c
    [...]
  end
  % equations here
end
```

Defining Component Equations

In this section...
“Equation Section Purpose” on page 2-20
“Equation Grammar” on page 2-21
“Specifying Mathematical Equality” on page 2-22
“Use of Relational Operators in Equations” on page 2-23
“Equation Dimensionality” on page 2-25
“Equation Continuity” on page 2-26
“Using Conditional Expressions in Equations” on page 2-27
“Using Intermediate Terms in Equations” on page 2-28
“Examples of Equations” on page 2-39

Equation Section Purpose



The equation section of a Simscape file follows the declaration and setup sections. It is executed throughout the simulation. The purpose of the equation section is to establish the mathematical relationships among a component’s variables, parameters, inputs, outputs, time and the time derivatives of each of these entities.

Equation Grammar

The following table describes the equation section grammar. Brackets {} indicate optional elements. The pipe | indicates a logical OR. Required characters and keywords are blue.

EquationSection	<code>equations</code> ExpressionList <code>end</code>
ExpressionList	EquationExpression { Separator EquationExpression }
EquationExpression	Expression CondExpression LetExpression
CondExpression	<code>if</code> Expression ExpressionList { <code>elseif</code> Expression ExpressionList } <code>else</code> ExpressionList <code>end</code>
LetExpression	<code>let</code> DeclarationList <code>in</code> ExpressionList <code>end</code>
DeclarationList	Declaration { Separator Declaration }
Declaration	LetValue = EquationExpression
LetValue	Identifier IdentifierList

where

Expression	MATLAB expression
Identifier	MATLAB identifier
IdentifierList	MATLAB identifier list in square brackets []
Separator	Comma, semicolon, or newline

Specifying Mathematical Equality

Simscape language stipulates semantically that all the equation expressions returned by the equation section of a Simscape file specify continuous mathematical equality between two expressions. Consider a simple example:

```
equations
    Expression1 == Expression2;
end
```

Here we have declared an equality between `Expression1` and `Expression2`. The left- and right-hand side expressions are any valid MATLAB expressions (see the next section for restrictions on using the relational operators: `==`, `<`, `>`, `<=`, `>=`, `~=`, `&&`, `|`). The equation expressions may be constructed from any of the identifiers defined in the model declaration.

The equation is defined with the `==` operator. This means that the equation does not represent assignment but rather a symmetric mathematical relationship between the left- and right-hand operands. Because `==` is symmetric, the left-hand operand is not restricted to just a variable. For example:

```
component MyComponent
    [...]
    variables
        a = 1;
        b = 1;
        c = 1;
    end
    equations
        a + b == c;
    end
end
```

The following example is mathematically equivalent to the previous example:

```
component MyComponent
    [...]
    variables
        a = 1;
        b = 1;
```

```

    c = 1;
end
equations
    0 == c - a - b;
end
end

```

Note Equation expressions must be terminated with a semicolon or a newline. Unlike MATLAB, the absence of a semicolon makes no difference. In any case, Simscape language does not display the result as it evaluates the equation.

Use of Relational Operators in Equations

In the previous section we discussed how `==` is used to declare mathematical equalities. In MATLAB, however, `==` yields an expression like any other operator. For example:

```
(a == b) * c;
```

where `a`, `b`, and `c` represent scalar double values, is a legal MATLAB expression. This would mean, take the `logical` value generated by testing `a`'s equivalence to `b`, coerce this value to a `double` and multiply by `c`. If `a` is the same as `b`, then this expression will return `c`. Otherwise, it will return 0.

On the other hand, in MATLAB we can use `==` twice to build an expression:

```
a == b == c;
```

This expression is ambiguous, but MATLAB makes `==` and other relational operators left associative, so this expression is treated as:

```
(a == b) == c;
```

The subtle difference between `(a == b) == c` and `a == (b == c)` can be significant in MATLAB, but is even more significant in an equation. Because the use of `==` is significant in the Simscape language, and to avoid ambiguity, the following syntax:

```
component MyComponent
```

```
[...]  
equations  
  a == b == c;  
end  
end
```

is illegal in the Simscape language. You must explicitly associate top-level occurrences of relational operators. Either

```
component MyComponent  
  [...]  
  equations  
    (a == b) == c;  
  end  
end
```

or

```
component MyComponent  
  [...]  
  equations  
    a == (b == c);  
  end  
end
```

are legal. In either case, the quantity in the parentheses is equated to the quantity on the other side of the equation.

With the exception of the top-level use of the == operator, == and other relational operators are left associative. For example:

```
component MyComponent  
  [...]  
  parameters  
    a = 1;  
    b = 1;  
    c = false;  
  end  
  variables  
    d = 1;
```

```

    end
  equations
    (a == b == c) == d;
  end
end

```

is legal and interpreted as:

```

component MyComponent
  [...]
  parameters
    a = 1;
    b = 1;
    c = false;
  end
  variables
    d = 1;
  end
  equations
    ((a == b) == c) == d;
  end
end

```

Equation Dimensionality

The expressions on either side of the == operator need not be scalar expressions. They must be either the same size or one must be scalar. For example:

```

equations
  [...]
  <3x3 Expression> == <3x3 Expression>;
  [...]
end

```

is legal and introduces 9 scalar equations. The equation expression:

```

equations
  [...]
  <1x1 Expression> == <3x3 Expression>;
  [...]

```

```
end
```

is also legal. Here, the left-hand side of the equation is expanded, via scalar expansion, into the same expression replicated into a 3x3 matrix. This equation expression also introduces 9 scalar equations.

However, the equation expression:

```
equations
  [...]
  <2x3 Expression> == <3x2 Expression>;
  [...]
end
```

is illegal because the sizes of the expressions on the left- and right-hand side are different.

Equation Continuity

The equation section is evaluated in continuous time. Some of the values that are accessible in the equation section are themselves piecewise continuous, that is, they change continuously in time. These values are:

- variables
- inputs
- outputs
- time

Piecewise continuous indicates that values are continuous over compact time intervals but may change value at certain instances. The following values are continuous, but not time-varying:

- parameters
- constants

Time-varying countable values, for example, integer or logical, are never continuous.

Continuity is propagated like a data type. It is propagated through continuous functions (see “Supported Functions”).

Using Conditional Expressions in Equations

You can specify conditional equations by using `if` statements.

```
equations
  [...]
  if Expression
    [...]
  elseif Expression
    [...]
  else
    [...]
  end
  [...]
end
```

Each `[...]` section may contain one or more equation expressions.

Note The total number of equation expressions, their dimensionality, and their order must be the same for every branch of the `if-elseif-else` statement.

You can nest `if` statements, for example:

```
equations
  [...]
  if Expression
    [...]
    if Expression
      [...]
    else
      [...]
    end
  else
    [...]
  end
```

```
end
[... ]
end
```

Every `if` requires an `else`.

Example

For a component where x and y are declared as 1x1 variables, specify the following piecewise equation:

$$y = \begin{cases} x & \text{for } -1 \leq x \leq 1 \\ x^2 & \text{otherwise} \end{cases}$$

This equation, written in the Simscape language, would look like:

```
equations
  if x >= -1 && x <= 1
    y == x;
  else
    y == x^2;
  end
end
```

Another way to write this equation in the Simscape language is:

```
equations
  y == if x>=-1 && x<=1, x else x^2 end
end
```

Using Intermediate Terms in Equations

- “Why Use Intermediate Terms?” on page 2-29
- “Syntax Rules” on page 2-30
- “Nested `let` Expressions” on page 2-33
- “Conditional `let` Expressions” on page 2-35
- “Identifier List in the Declarative Clause” on page 2-37

Why Use Intermediate Terms?

Textbooks often define certain equation terms in separate equations, and then substitute these intermediate equations into the main one. For example, for fully developed flow in ducts, the Darcy friction factor can be used to compute pressure loss:

$$P = \frac{f \cdot L \cdot \rho \cdot V^2}{2D}$$

where P is pressure, f is the Darcy friction factor, L is length, ρ is density, V is flow velocity, and D is hydraulic area.

These terms are further defined by:

$$f = \frac{0.316}{\text{Re}^{1/4}}$$

$$\text{Re} = \frac{D \cdot V}{\nu}$$

$$D = \sqrt{\frac{4A}{\pi}}$$

$$V = \frac{q}{A}$$

where Re is the Reynolds number, A is the area, q is volumetric flow rate, and ν is the kinematic viscosity.

In Simscape language, you can define intermediate terms and use them in one or more equations by using the `let` expressions. The following example shows the same equations written out in Simscape language:

```
component MyComponent
[... ]
parameters
    L = { 1, 'm' }; % length
    rho = { 1e3, 'kg/m^3' }; % density
```

```
        nu = { 1e-6, 'm^2/s' }; % kinematic viscosity
    end
    variables
        p = { 0, 'Pa' }; % pressure
        q = { 0, 'm^3/s' }; % volumetric flow rate
        A = { 0, 'm^2' }; % area
    end
    equations
        let
            f = 0.316 / Re_d^0.25; % Darcy friction factor
            Re_d = D_h * V / nu; % Reynolds number
            D_h = sqrt( 4.0 * A / pi ); % hydraulic area
            V = q / A; % flow velocity
        in
            p == f * L * rho * V^2 / (2 * D_h); % final equation
        end
    end
end
```

After substitution of all intermediate terms, the final equation becomes:

```
p==0.316/(sqrt(4.0 * A / pi) * q / A / nu)^0.25 * L * rho * (q / A)^2 / (2 * sqrt(4.0 * A / pi));
```

Syntax Rules

A `let` expression consists of two clauses, the declaration clause and the expression clause.

```
equations
    [...]
    let
        declaration clause
    in
        expression clause
    end
    [...]
end
```

The declaration clause assigns an identifier, or set of identifiers, on the left-hand side of the equal sign (=) to an equation expression on the right-hand side of the equal sign:

```
LetValue = EquationExpression
```

The expression clause defines the scope of the substitution. It starts with the keyword `in`, and may contain one or more equation expressions. All the expressions assigned to the identifiers in the declaration clause are substituted into the equations in the expression clause during parsing.

Note The end keyword is required at the end of a `let-in-end` statement.

Here is a simple example:

```
component MyComponent
  [...]
  variables
    x = 0;
    y = 0;
  end
  equations
    let
      z = y + 1;
    in
      x == z;
    end
  end
end
```

In this example, the declaration clause of the `let` expression sets the value of the identifier `z` to be the expression `y + 1`. Thus, substituting `y + 1` for `z` in the expression clause in the `let` statement, the code above is equivalent to:

```
component MyComponent
  [...]
  variables
    x = 0;
    y = 0;
```

```
end
equations
  x == y + 1;
end
end
end
```

There may be multiple declarations in the declaration clause. These declarations are order independent. The identifiers declared in one declaration may be referred to by the expressions for identifiers in other declarations in the same declaration clause. Thus, in the code example shown in the previous section, the identifier `Re_d` (Reynolds number) is used in the expression declaring the identifier `f` (Darcy friction factor). The only requirement is that the expression references are acyclic.

The expression clause of a `let` expression defines the scope of the substitution for the declaration clause. Other equations, that do not require these substitutions, may appear in the equation section outside of the expression clause. In the following example, the equation section contains the equation expression `c == b + 2` outside the scope of the `let` expression before it.

```
component MyComponent
[... ]
variables
  a = 0;
  b = 0;
  c = 0;
end
equations
  let
    x = a + 1;
  in
    b == x;
  end
  c == b + 2;
end
end
```

These expressions are treated as peers. They are order independent, so this example is equivalent to

```
component MyComponent
[...]  
variables  
  a = 0;  
  b = 0;  
  c = 0;  
end  
equations  
  c == b + 2;  
  let  
    x = a + 1;  
  in  
    b == x;  
  end  
end  
end
```

and, after the substitution, to

```
component MyComponent
[...]  
variables  
  a = 0;  
  b = 0;  
  c = 0;  
end  
equations  
  b == a + 1;  
  c == b + 2;  
end  
end
```

Nested let Expressions

You can nest let expressions, for example:

```
component MyComponent
[...]  
variables  
  a = 0;
```

```
        b = 0;
        c = 0;
    end
    equations
        let
            w = a + 1;
        in
            let
                z = w + 1;
            in
                b == z;
                c == w;
            end
        end
    end
end
```

In case of nesting, substitutions are performed based on both of the declaration clauses. After the substitutions, the code above becomes:

```
component MyComponent
[... ]
variables
    a = 0;
    b = 0;
    c = 0;
end
equations
    b == a + 1 + 1;
    c == a + 1;
end
end
```

The innermost declarations take precedence. The following example illustrates a nested `let` expression where the inner declaration clause overrides the value declared in the outer one:

```
component MyComponent
[... ]
variables
```



```
    a = 0;
    b = 0;
end
equations
  let
    w = a + 1;
  in
    let
      w = a + 2;
    in
      b == w;
    end
  end
end
end
```

Performing substitution on this example yields:

```
component MyComponent
[... ]
variables
  a = 0;
  b = 0;
end
equations
  b == a + 2;
end
end
```

Conditional let Expressions

You can use `if` statements within both declarative and expression clause of `let` expressions, for example:

```
component MyComponent
[... ]
variables
  a = 0;
  b = 0;
  c = 0;
```

```
end
equations
  let
    x = if a < 0, a else b end;
  in
    c == x;
  end
end
end
```

Here x is declared as the conditional expression based on $a < 0$. Performing substitution on this example yields:

```
component MyComponent
[...]
```

```
variables
  a = 0;
  b = 0;
  c = 0;
end
equations
  c == if a < 0, a else b end;
end
end
```

The next example illustrates how you can use `let` expressions within conditional expressions. The two `let` expressions on either side of the conditional expression are independent:

```
component MyComponent
[...]
```

```
variables
  a = 0;
  b = 0;
  c = 0;
end
equations
  if a < 0
    let
      z = b + 1;
```

```
        in
          c == z;
        end
      else
        let
          z = b + 2;
        in
          c == z;
        end
      end
    end
  end
end
```

This code is equivalent to:

```
component MyComponent
[... ]
variables
  a = 0;
  b = 0;
  c = 0;
end
equations
  if a < 0
    c == b + 1;
  else
    c == b + 2;
  end
end
end
```

Identifier List in the Declarative Clause

This example shows using an identifier list, rather than a single identifier, in the declarative clause of a `let` expression:

```
component MyComponent
[... ]
variables
```

```
    a = 0;
    b = 0;
    c = 0;
    d = 0;
end
equations
  let
    [x, y] = if a < 0, a; -a else -b; b end;
  in
    c == x;
    d == y;
  end
end
end
```

Here x and y are declared as the conditional expression based on $a < 0$. Notice that each side of the `if` statement defines a list of two expressions. A first semantic translation of this example separates the `if` statement into

```
if a < 0, a; -a else -b; b end =>
{ if a < 0, a else -b end; if a < 0, -a else b end }
```

then the second semantic translation becomes

```
[x, y] = { if a < 0, a else -b end; if a < 0, -a else b end } =>
x = if a < 0, a else -b end; y = if a < 0, -a else b end;
```

and the final substitution on this example yields:

```
component MyComponent
[... ]
variables
  a = 0;
  b = 0;
  c = 0;
  d = 0;
end
equations
  c == if a < 0, a else -b end;
  d == if a < 0, -a else b end;
end
```

end

Examples of Equations

- “Simple Algebraic System” on page 2-39
- “Using Simulation Time in Equations” on page 2-39

Simple Algebraic System

This example shows implementation for a simple algebraic system:

$$y = x^2$$

$$x = 2y + 1$$

The Simscape file looks as follows:

```
component MyAlgebraicSystem
  variables
    x = 0;
    y = 0;
  end
  equations
    y == x^2;          % y = x^2
    x == 2 * y + 1;  % x = 2 * y + 1
  end
end
```

Using Simulation Time in Equations

You can access global simulation time from the equation section using the `time` function. `time` returns the simulation time in seconds.

The following example illustrates $y = \sin(\omega t)$, where t is simulation time:

```
component
  parameters
    w = { 1, '1/s' } % omega
  end
```

```
    outputs
      y = 0;
    end
    equations
      y == sin( w * time );
    end
  end
```

Putting It Together — Complete Component Examples

In this section...

“Mechanical Component Example — Spring” on page 2-41

“Electrical Component Example — Ideal Capacitor” on page 2-42

“No-Flow Component Example — Voltage Sensor” on page 2-44

“Grounding Component Example — Electrical Reference” on page 2-45

Mechanical Component Example — Spring

The following file, `spring.ssc`, implements a component called `spring`.

The declaration section of the component contains:

- Two rotational nodes, `r` and `c` (for rod and case, respectively)
- Parameter `k`, with a default value of $10 \text{ N}\cdot\text{m}/\text{rad}$, specifying the spring rate
- Through and Across variables, torque `t` and angular velocity `w`, to be connected to the rotational domain at setup
- Internal variable `theta`, with a default value of 0 rad , specifying relative angle, that is, deformation of the spring

The setup section of the component performs the following:

- Checks that the spring rate constant is nonnegative
- Establishes relationships between the component variables and nodes (and therefore domain variables) using `through` and `across` functions

The equation section of the component contains two equations that define the spring action:

- $t = k * \text{theta}$, that is, torque equals spring deformation times spring rate
- $w = \text{theta}'$, that is, angular velocity equals time derivative of spring deformation

```
component spring
```

```
nodes
    r = foundation.mechanical.rotational.rotational;
    c = foundation.mechanical.rotational.rotational;
end
parameters
    k = { 10, 'N*m/rad' };
end
variables
    theta = { 0, 'rad' };
    t = { 0, 'N*m' }; % torque through
    w = { 0, 'rad/s' }; % velocity across
end
function setup
    if k < 0
        error( 'Spring rate must be greater than zero' );
    end
    through( t, r.t, c.t ); % torque through from node r to node c
    across( w, r.w, c.w ); % velocity across from r to c
end
equations
    t == k * theta;
    w == theta.der;
end
end
```

Electrical Component Example – Ideal Capacitor

The following file, `ideal_capacitor.ssc`, implements a component called `ideal_capacitor`.

The declaration section of the component contains:

- Two electrical nodes, `p` and `n` (for + and – terminals, respectively)
- Two parameters: `C`, with a default value of 1 F, specifying the capacitance, and `V0`, with a default value of 0 V, specifying the initial voltage
- Through and Across variables, current `i` and voltage `v`, to be connected to the electrical domain at setup

The setup section of the component performs the following:

- Checks that the capacitance is nonnegative
- Establishes relationships between the component variables and nodes (and therefore domain variables) using through and across functions

The equation section of the component contains the equation that defines the capacitor action:

- $I = C \cdot dV/dt$, that is, output current equals capacitance multiplied by the time derivative of the input voltage

```

component ideal_capacitor
% Ideal Capacitor
% Models an ideal (lossless) capacitor. The output current I is related
% to the input voltage V by  $I = C \cdot dV/dt$  where C is the capacitance.

nodes
    p = foundation.electrical.electrical; % +:top
    n = foundation.electrical.electrical; % -:bottom
end
parameters
    C = { 1, 'F' }; % Capacitance
    V0 = { 0, 'V' }; % Initial voltage
end
variables
    i = { 0, 'A' }; % Current through variable
    v = { 0, 'V' }; % Voltage across variable
end
function setup
    if C <= { 0, 'F' }
        error( 'Capacitance must be greater than zero' )
    end
    through( i, p.i, n.i ); % Through variable i from node p to node n
    across( v, p.v, n.v ); % Across variable v from p to n
    v = V0; % v(t=0) == V0
end
equations
    i == C*v.der; % Equation
end
end

```

No-Flow Component Example – Voltage Sensor

The following file, `voltage_sensor.ssc`, implements a component called `voltage_sensor`. An ideal voltage sensor has a very large resistance, so there is no current flow through the sensor. Therefore, declaring a Through variable, as well as writing setup and equation statements for it, is unnecessary.

The declaration section of the component contains:

- Two electrical nodes, `p` and `n` (for + and – terminals, respectively)
- An Across variable, voltage `v1`, to be connected to the electrical domain at setup. Note that a Through variable (current) is not declared.

The setup section of the component performs the following:

- Establishes the relationship between the component variable, voltage `v1`, and nodes (and therefore domain variables) using the `across` function. Again, note that there is no `through` function at setup.

The equation section of the component contains the equation that defines the voltage sensor action:

- `V == v1`, that is, output voltage equals the voltage across the sensor nodes

```
component voltage_sensor
% Voltage Sensor
% The block represents an ideal voltage sensor. There is no current
% flowing through the component, therefore it is unnecessary to
% declare a Through variable (i1), setup a 'through' function, or
% create an equation statement (such as i1 == 0).
%
% Connection V is a physical signal port that outputs voltage value.

outputs
    V = { 0.0, 'V' }; % V:bottom
end

nodes
```

```

    p = foundation.electrical.electrical; % +:top
    n = foundation.electrical.electrical; % -:bottom
end

variables
    v1 = { 0, 'V' };
end

function setup
    across( v1, p.v, n.v );
end

equations
    V == v1;
end

end

```

Grounding Component Example — Electrical Reference

The following file, `elec_reference.ssc`, implements a component called `elec_reference`. This component provides an electrical ground to a circuit. It has one node, where the voltage equals zero. It also declares a current variable, makes it incident to the component node using the `through` function, and does not specify any value for it in the equation section. Therefore, it can take on any value and handle the current flowing into or out of the reference node.

The declaration section of the component contains:

- One electrical node, `V`
- A Through variable, current `i`, to be connected to the electrical domain at `setup`. Note that there is no need to declare an Across variable (voltage) because this is a grounding component.

The setup section of the component performs the following:

- Uses the `through` function to establish the relationship between the component variable, current `i`, and the domain variables. The second

argument is associated with the component node, `V`. The third argument is replaced with `[]`, to indicate the reference node.

There is no need for the `across` function at setup.

The equation section of the component contains the equation that defines the grounding action:

- `V.v == 0`, that is, voltage at the node equals zero

```
component elec_reference
% Electrical Reference
% Electrical reference port. A model must contain at least one
% electrical reference port (electrical ground).

nodes
    V = foundation.electrical.electrical; % :top
end

variables
    i = { 0, 'A' };
end

function setup
    through( i, V.i, [] );
end

equations
    V.v == 0;
end

end
```

Working with Domain Parameters

In this section...

“Propagation of Domain Parameters” on page 2-47

“Source Components” on page 2-48

“Propagating Components” on page 2-48

“Blocking Components” on page 2-49

“Example of Using Domain Parameters” on page 2-49

Propagation of Domain Parameters

The purpose of domain parameters is to propagate the same parameter value to all or some of the components connected to the domain. For example, this hydraulic domain contains one Across variable, p , one Through variable, q , and one parameter, t .

```
domain t_hyd
variables
  p = { 1e6, 'Pa' }; % pressure
end
variables(Balancing = true)
  q = { 1e-3, 'm^3/s' }; % flow rate
end
parameters
  t = { 303, 'K' }; % fluid temperature
end
end
```

All components with nodes connected to this domain will have access to the fluid temperature parameter t . The component examples in the following sections assume that this domain file, `t_hyd.ssc`, is located in a package named `+THyd`.

When dealing with domain parameters, there are three different types of components. There are some components that will provide the domain parameters to the larger model, there are some that simply propagate the parameters, and there are some that do not propagate parameters.

The behavior of the component is specified by the component attribute `Propagation`. The `Propagation` attribute may be set to one of three options: `propagates`, `source`, or `blocks`. For more information, see “Attribute Lists” on page 2-56.

Source Components

The source setting is used for components that provide parameters to other parts of the model, source components. The following is an example of a source component, connected to the hydraulic domain `t_hyd`, defined in “Propagation of Domain Parameters” on page 2-47. This component provides the value of the temperature parameter to the rest of the model.

```
component ( Propagation = source ) hyd_temp
% Hydraulic Temperature
% Provide hydraulic temperature to the rest of the model
parameters
    t = { 333, 'K' }; % Fluid temperature
end
nodes
    a = THyd.t_hyd; % t_hyd node
end
function setup
    a.t = t; % set temperature at node to temperature parameter
end
end
```

When you generate a Simscape block from this component file, the block dialog box will have a parameter labelled **Fluid temperature**. You can then use it to enter the temperature value for the hydraulic fluid used in the model.

If a component is specified as a source component and does not set all of the domain parameters of all of its public nodes, an error will result.

Propagating Components

The default setting for the `Propagation` component attribute is `propagates`. Most components use this setting. If a component is configured to propagate its domain parameters, then all public nodes connected to this domain have

the same set of domain parameters. These parameters are accessible in the setup and equation sections of the component file.

The following is an example of a propagating component `h_temp_sensor`, connected to the hydraulic domain `t_hyd`, defined in “Propagation of Domain Parameters” on page 2-47. It outputs the fluid temperature as a physical signal `T`. This example shows how you can access domain parameters in the equation section of a component.

```

component h_temp_sensor
% Hydraulic Temperature Sensor
%     Measure hydraulic temperature
outputs
    T = { 0, 'K' }; % T:right
end
nodes
    a = THyd.t_hyd; % t_hyd node
end
equations
    T == a.t; % access parameter directly from node in equations
end
end

```

Blocking Components

Blocking components are those components that do not propagate domain parameters. These components have their `Propagation` attribute set to `blocks`. It is illegal to access domain parameters in blocking components.

Example of Using Domain Parameters

The following example shows how you can test propagation of domain parameters by putting together a simple circuit. In this example, you will:

- Create the necessary domain and component files and organize them in a package. For more information, see “Organizing Your Simscape Files” on page 3-3.
- Build a custom block library based on these Simscape files. For more information, see “Converting Your Simscape Files” on page 3-4.

- Use these custom blocks to build a model and test propagation of domain parameters.

To complete the tasks listed above, follow these steps:

- 1** In a directory located on the MATLAB path, create a directory called +THyd. This is your package directory, where you store all Simscape files created in the following steps.
- 2** Create the domain file `t_hyd.ssc`, as described in “Propagation of Domain Parameters” on page 2-47.

```
domain t_hyd
variables
    p = { 1e6, 'Pa' }; % pressure
end
variables(Balancing = true)
    q = { 1e-3, 'm^3/s' }; % flow rate
end
parameters
    t = { 303, 'K' }; % fluid temperature
end
end
```

- 3** Create the component file `hyd_temp.ssc`, as described in “Source Components” on page 2-48. This component provides the value of the temperature parameter to the rest of the model.

```
component ( Propagation = source ) hyd_temp
% Hydraulic Temperature
% Provide hydraulic temperature to the rest of the model
parameters
    t = { 333, 'K' }; % Fluid temperature
end
nodes
    a = THyd.t_hyd; % t_hyd node
end
function setup
    a.t = t; % set temperature at node to temperature parameter
end
end
```


- 4** Create the component file `h_temp_sensor.ssc`, as described in “Propagating Components” on page 2-48. This component measures the value of the temperature parameter and outputs it as a physical signal.

```

component h_temp_sensor
% Hydraulic Temperature Sensor
%     Measure hydraulic temperature
outputs
    T = { 0, 'K' }; % T:right
end
nodes
    a = THyd.t_hyd; % t_hyd node
end
equations
    T == a.t; % access parameter directly from node in equations
end
end

```

- 5** In order to create a working circuit, you will need a reference block corresponding to the domain type, as described in “Grounding Rules”. Create a reference component for your `t_hyd` domain, as follows (name the component `h_temp_ref.ssc`):

```

component h_temp_ref
% Hydraulic Temperature Reference
%     Provide reference for thermohydraulic circuits
nodes
    a = THyd.t_hyd; % t_hyd node
end
variables
    q = { 0, 'm^3/s' };
end
function setup
    through( q, a.q, [] );
end
equations
    a.p == 0;
end
end

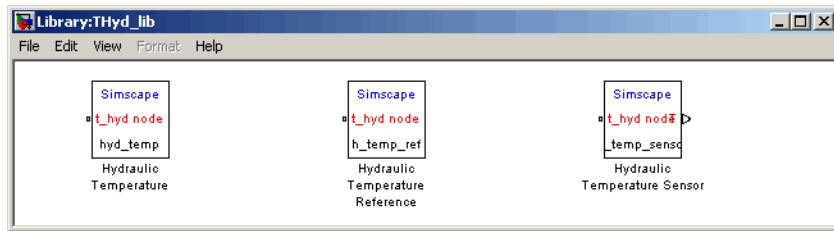
```

- 6** You can optionally define other components referencing the `t_hyd` domain, but this basic set of components is enough to create a working circuit. Now you need to build a custom block library based on these Simscape files. To do this, at the MATLAB command prompt, type:

```
ssc_build THyd;
```

- 7** This command generates a file called `THyd_lib.mdl` in the directory that contains your `+THyd` package. Before using this library, restart MATLAB to register the new domain. Then open the custom library by typing:

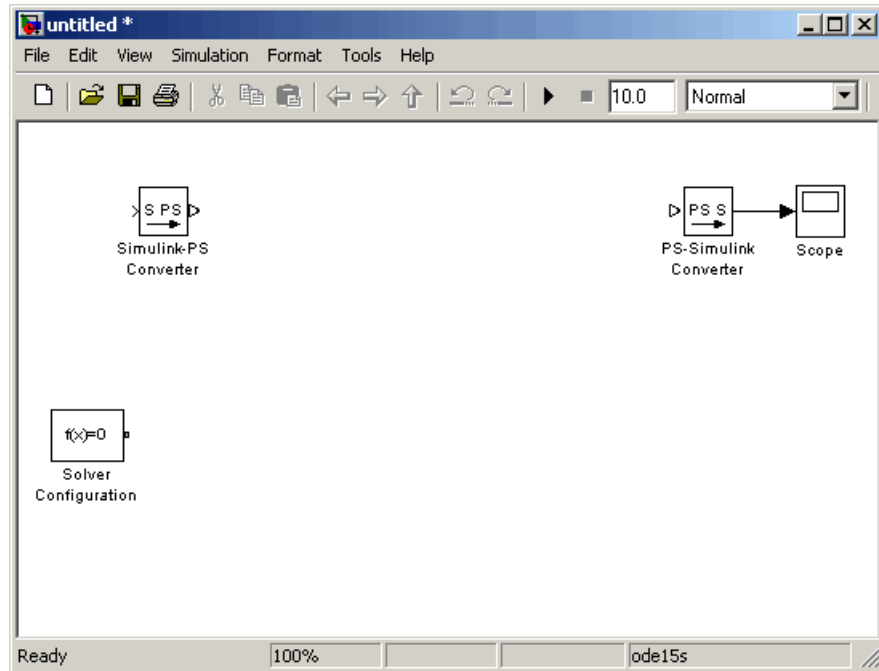
```
THyd_lib
```



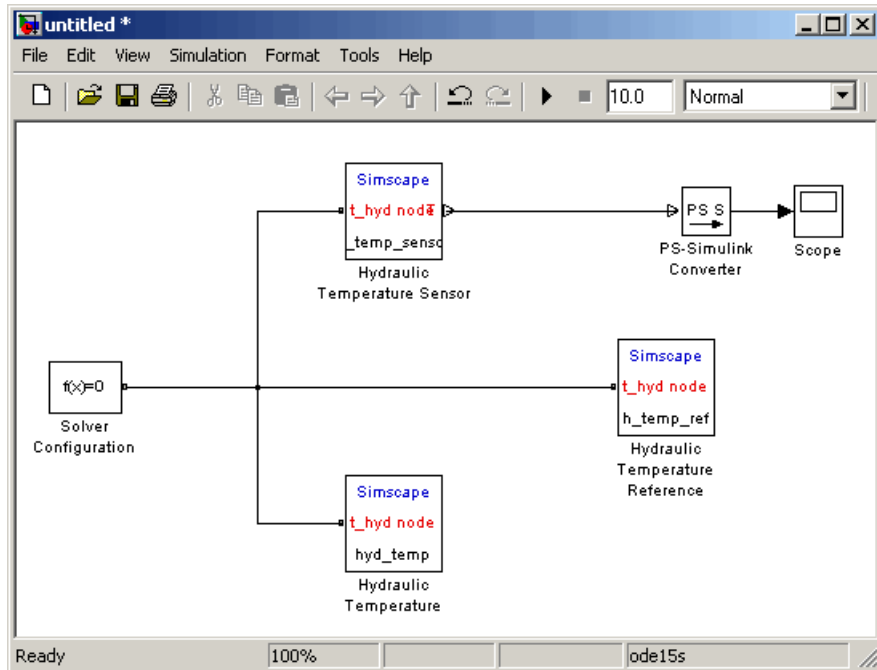
- 8** Create a new Simscape model. To do this, type:

```
ssc_new
```

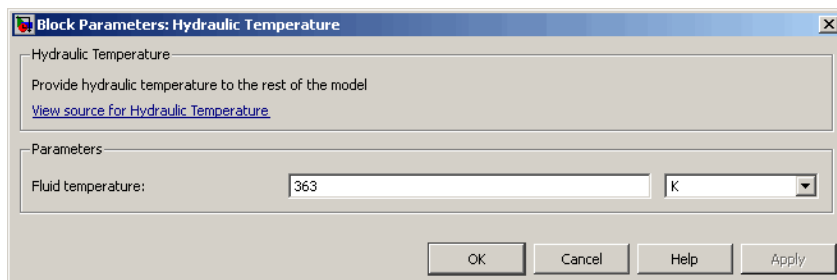
This command creates a new model, prepopulated with the following blocks:



- 9 Delete the Simulink-PS Converter block, because our model is not going to have any Simulink input signals.
- 10 Drag the Hydraulic Temperature, Hydraulic Temperature Sensor, and Hydraulic Temperature Reference blocks from THyd_lib.mdl and connect them as follows:



- 11 Simulate the model and notice that the scope displays the value of the domain temperature parameter, as it is defined in the `hyd_temp.ssc` file, 333 K.
- 12 Double-click the Hydraulic Temperature block. Change the value of the **Fluid temperature** parameter to 363 K.



- 13** Simulate the model again and notice that the scope now displays the new value of the domain temperature parameter.

Attribute Lists

In this section...
“Attribute Types” on page 2-56
“Model Attributes” on page 2-56
“Member Attributes” on page 2-57

Attribute Types

The attributes appear in an `AttributeList`, which is a comma separated list of pairs, as defined in the MATLAB class system grammar. Simscape file grammar allows attribute lists to appear in two situations:

Model	<pre>ModelClass { (AttributeList) Space } Identifier { < Identifier } Separator DeclarationSection Separator { SetupSection Separator } { EquationSection Separator } end</pre>
DeclarationBlock	<pre>MemberClass { (AttributeList) } Separator Identifier = Expression Separator { Identifier = Expression Separator } end</pre>

Therefore, Simscape language distinguishes between two types of attributes: model attributes and member attributes.

Model Attributes

Model attributes are applicable only to model type component.

Attribute	Values	Default	Model Classes	Description
Propagation	propagates source blocks	propagates	component	Defines the domain data propagation of the component. See “Propagation of Domain Parameters” on page 2-47.
Hidden	true false	false	component	Defines the visibility of the entire component. This dictates whether the component shows up in a generated library or report.

Component model attributes apply to the entire model. For example:

```
component MyParameterSource ( Propagation = source )
  % component model goes here
end
```

Here, Propagation is a model attribute.

Member Attributes

Member attributes apply to a whole DeclarationBlock.

Attribute	Values	Default	Member Classes	Description
Access	public private protected	public	all	Defines the read and write access of members. Private members are only accessible to the instance of the component model and not to external clients.
Hidden	true false	false	all	Sets the visibility of the member in the user interface.

Attribute	Values	Default	Member Classes	Description
Balancing	true false	false	variables	If set to true, declares Through variables for a domain. You can set this attribute to true only for model type domain. See “Declaring Through and Across Variables for a Domain” on page 2-6.
Conversion	absolute relative	absolute	parameters	Defines how the parameter units are converted for use in the setup and equation sections. See “Specifying Parameter Units” on page 2-8.

The attribute list for the DeclarationBlock appears after MemberClass keyword. For example:

```
parameters ( Access = public, Hidden = true )
    % parameters go here
end
```

Here, all parameters in the declaration block are externally writable, but they will not appear in the block dialog box.

Subclassing and Inheritance

Subclassing allows you to build component models based on other component models by extension. Subclassing applies only to component models, not domain models. The syntax for subclassing is based on the MATLAB class system syntax for subclassing using the < symbol on the declaration line of the component model:

```
component MyExtendedComponent < PackageName.MyBaseComponent
    % component implementation here
end
```

By subclassing, the subclass inherits all of the members (parameters, variables, nodes, inputs and outputs) from the base class and can add members of its own. When using the subclass as an external client, all **public** members of the base class are available. All **public** and **protected** members of the base class are available to the setup and equation functions of the subclass. The subclass may not declare a member with the same identifier as a **public** or **protected** member of the base class.

The setup function of the base class is executed before the setup function of the subclass. The equations of both the subclass and the base class are included in the overall system of equations. For example, you can define the base class `ElectricalBranch.ssc` as:

```
component ElectricalBranch
% Defines an electrical branch with positive
% and negative external nodes. Also defines internal
% current and voltage.
nodes
    p = foundation.electrical.electrical;
    n = foundation.electrical.electrical;
end
variables
    i = { 0, 'A' };
    v = { 0, 'V' };
end
function setup
    across( v, p.v, n.v );
    through( i, p.i, n.i );
```

```
    end  
end
```

If, for example, your base class resides in a package named `+MyElectrical`, then you can define the subclass component `Capacitor.ssc` as follows:

```
component Capacitor < MyElectrical.ElectricalBranch  
  parameters  
    c = { 1, 'F' };  
  end  
  function setup  
    if c <= 0  
      error( 'Capacitance must be greater than zero' );  
    end  
  end  
  equations  
    i == c * v.der;  
  end  
end
```

Using Simscape Files in Block Diagrams

- “Adding Custom Block Libraries Generated from Simscape Component Files” on page 3-2
- “Customizing the Block Name and Appearance” on page 3-11
- “Checking File and Model Dependencies” on page 3-22
- “Case Study — Creating a Basic Custom Block Library” on page 3-26
- “Case Study — Creating an Electrochemical Library” on page 3-33

Adding Custom Block Libraries Generated from Simscape Component Files

In this section...

“Workflow Overview” on page 3-2

“Organizing Your Simscape Files” on page 3-3

“Using Source Protection for Simscape Files” on page 3-3

“Converting Your Simscape Files” on page 3-4

“Customizing the Library Name and Appearance” on page 3-5

“Customizing the Library Icon” on page 3-7

“Example — Creating and Customizing Block Libraries” on page 3-8

“Specifics of Using Customized Domains” on page 3-9

Workflow Overview

After you have created the textual component files, you need to convert them into Simscape blocks to be able to use them in block diagrams. This process involves:

- 1** Organizing your Simscape files. Simscape files must be saved in package directories. The package hierarchy determines the resulting library structure.
- 2** Optional source protection. If you want to share your models with customers without disclosing the component or domain source, you can generate Simscape protected files and share those.
- 3** Building the custom block library. You can use either the regular Simscape source files or Simscape protected files to do this. Each top-level package generates a separate custom Simscape block library.

Once you generate the custom Simscape library, you can open it and drag the customized blocks from it into your models.

Organizing Your Simscape Files

Simscape files must be saved in package directories. For more information on package directories, see “Scoping Classes with Packages” in the *MATLAB Classes and Object-Oriented Programming* documentation. The important points are:

- The package directory name must begin with a + character.
- The rest of the package directory name (without the + character) must be a valid MATLAB identifier.
- The package directory’s parent directory must be on the MATLAB path.

Each package where you store your Simscape files generates a separate custom block library.

Package directories may be organized into subdirectories, with names also beginning with a + character. After you build a custom block library, each such subdirectory will appear as a sublibrary under the top-level custom library.

For example, you may have a top-level package directory, named `+SimscapeCustomBlocks`, and it has three subdirectories, `Electrical`, `Hydraulic`, and `Mechanical`, each containing Simscape files. The custom block library generated from this package will be called `SimscapeCustomBlocks_lib.mdl` and will have three corresponding sublibraries. For information on building custom block libraries, see “Converting Your Simscape Files” on page 3-4.

Using Source Protection for Simscape Files

If you need to protect your proprietary source code when sharing the Simscape files, use one of the following commands to generate Simscape protected files:

- `ssc_protect` — Protects individual files and directories. Once you encrypt the files, you can share them without disclosing the component or domain source. Use them, just as you would the Simscape source files, to build custom block libraries with the `ssc_build` command.
- `ssc_mirror` — Creates a protected copy of a whole package in a specified directory. Setting a flag lets you also build a custom block library from the protected files and place it in the mirror directory, thus eliminating the

need to run the `ssc_build` command. Use the `ssc_mirror` command to quickly prepare a whole package for sharing with your customers, without disclosing the component or domain source.

Unlike Simscape source files, which have the extension `.ssc`, Simscape protected files have the extension `.sscp` and are not humanly-readable. You can use them, just as the Simscape source files, to build custom block libraries. Protected files have to be organized in package directories, in the same way as the Simscape source files. For information on organizing your files, see “Organizing Your Simscape Files” on page 3-3. For information on building custom block libraries, see “Converting Your Simscape Files” on page 3-4.

Converting Your Simscape Files

After you have created the textual component files and organized them in package directories, you need to convert them into Simscape blocks to be able to use them in block diagrams. You do this by running the `ssc_build` command on the top-level package directory containing your Simscape files. The package may contain either the regular Simscape source files or Simscape protected files.

Note Before running the `ssc_build` command for the first time, you have to set up your compiler by running `mex -setup`. For more information, see “Building MEX-Files” in the *MATLAB External Interfaces* documentation.

For example, you may have a top-level package directory, where you store your Simscape files, named `+SimscapeCustomBlocks`. To generate a custom block library, at the MATLAB command prompt, type:

```
ssc_build SimscapeCustomBlocks;
```

Note The package directory name begins with a leading `+` character, whereas the argument to `ssc_build` must omit the `+` character.

This command generates a file called `SimscapeCustomBlocks_lib.mdl` in the parent directory of the top-level package (that is, in the same directory that

contains your +SimscapeCustomBlocks package). Because this directory is on the MATLAB path, you can open the library by typing its name at the MATLAB command prompt. In our example, type:

```
SimscapeCustomBlocks_lib
```

The model file generated by running the `ssc_build` command is the custom Simscape library containing all the sublibraries and blocks generated from the Simscape files located in the top-level package. Once you open the custom Simscape library, you can drag the customized blocks from it into your models.

Creating Sublibraries

Package directories may be organized into subdirectories, with names also beginning with a + character. After you run the `ssc_build` command, each such subdirectory will appear as a sublibrary under the top-level custom library. You can customize the name and appearance of sublibraries by using library configuration files.

Note When you add or modify component files in package subdirectories, you still run the `ssc_build` command on the top-level package directory. This updates all the sublibraries.

You may have more than one top-level package directory, that is, more than one package directory located in a directory on the MATLAB path. Each top-level package directory generates a separate top-level custom library.

Customizing the Library Name and Appearance

Package names must be valid MATLAB identifiers. The top-level package always generates a library model with the name `package_name_lib.mdl`. However, library configuration files let you provide descriptive library names and specify other customizations for sublibraries, generated from subdirectories in the package hierarchy.

A library configuration file must be located in the package directory and named `lib.m`.

Library configuration files are not required. You can choose to provide `lib.m` for some subpackages, all subpackages, or for none of the subpackages. If a subpackage does not contain a `lib.m` file, the sublibrary is built using the default values. The top-level package can also contain a `lib.m` file. Options such as library name, and other options that do not make sense for a top-level library, are ignored during build. However, having a file with the same name and options in the top-level package provides a uniform mechanism that lets you easily change the library hierarchy.

The following table describes the supported options. The only option that is required in a `lib.m` file is `Name`; others are optional.

Option	Usage	Description	Default	For Top-Level Package
Name	<code>libInfo.Name = <i>name</i></code>	<i>name</i> will be used as the name of the sublibrary (name of the Simulink subsystem corresponding to the sublibrary)	Package name	Ignored
Annotation	<code>libInfo.Annotation = <i>annotation</i></code>	<i>annotation</i> will be displayed as annotation when you open the sublibrary. It can be any text that you want to display in the sublibrary.	No annotation in the library	Used in annotation for top-level library
ShowIcon	<code>libInfo.ShowIcon = false</code>	If there is no library icon file <code>lib.img</code> , as described in “Customizing the Library Icon” on page 3-7, this option is ignored. If there is an icon file, you can choose to not use it by setting this option to <code>false</code> .	true	Ignored

Option	Usage	Description	Default	For Top-Level Package
ShowName	<code>libInfo.ShowName = true</code>	Allows you to configure whether the sublibrary name is shown in the parent library. If there is no library icon file, then the default library icon contains the library name, and showing it again is redundant. If you are using a library icon file, set <code>showName</code> to <code>true</code> to display the library name below the icon.	false	Ignored
Hidden	<code>libInfo.Hidden = true</code>	Allows you to configure whether the sublibrary is visible in the parent library. Use this option for a sublibrary containing blocks that you do not want to expose, for example, those kept for compatibility reasons.	false	Ignored

Customizing the Library Icon

If a subpackage contains a file named `lib.img`, where *img* is one of the supported image file formats (such as `jpg`, `bmp`, or `png`), then that image file is used for the icon representing this sublibrary in the parent library. The icon file (`lib.img`) and customization file (`lib.m`) are independent, you can provide one or the other, both, or none.

The following image file formats are supported:

- `jpg`
- `bmp`
- `png`

If there are multiple image files, the formats take precedence in the order listed above. For example, if a subpackage contains both `lib.jpg` and `lib.bmp`, `lib.jpg` is the image that will appear in the parent library.

You can turn off customizing the library icon by setting `showIcon` to `false` in the library customization file `lib.m`. In this case, the default library icon will be used. For more information, see “Customizing the Library Name and Appearance” on page 3-5.

Example – Creating and Customizing Block Libraries

Consider the following directory structure:

```
- +MySimscapeLibrary
  |-- +MechanicalElements
  |   |-- lib.m
  |   |-- lib.jpg
  |   |-- inertia.ssc
  |   |-- spring.ssc
  |-- +ElectricalElements
  |   |-- ...
  |-- +HydraulicElements
  |   |-- ...
```

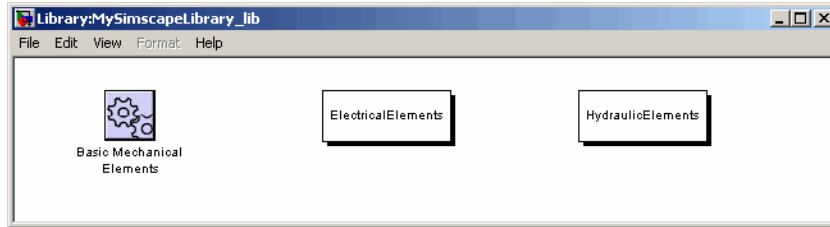
This means that you have a top-level package called `+MySimscapeLibrary`, which contains three subpackages, `+MechanicalElements`, `+ElectricalElements`, and `+HydraulicElements`. The `+MechanicalElements` package contains two component files, `inertia.ssc` and `spring.ssc`, a library icon file `lib.jpg`, and the following library configuration file `lib.m`:

```
function lib ( libInfo )
    libInfo.Name = 'Basic Mechanical Elements';
    libInfo.Annotation = sprintf('This library contains basic mechanical elements');
    libInfo.ShowName = true;
```

When you run

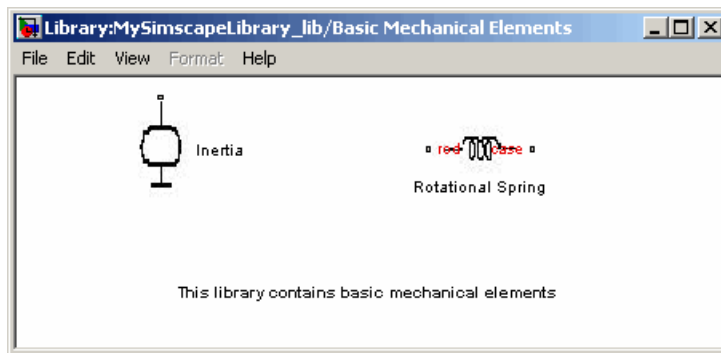
```
ssc_build MySimscapeLibrary;
```

the top-level package generates a library model called `MySimscapeLibrary_lib`, as follows:



Notice that the sublibrary generated from the +MechanicalElements package is presented in its parent library with a customized icon and name (Basic Mechanical Elements).

If you double-click the Basic Mechanical Elements sublibrary, it opens as follows:



Specifics of Using Customized Domains

If you define your own physical domains and use them in your custom blocks, several rules apply.

- If the domain file resides in a different package from the components that use it, build the domain package first.
- If you have opened the Library Browser, or a Simulink library or model, before building the package containing the domain, you have to restart MATLAB before you can use the components referencing this domain in your block diagrams.

Otherwise, all the rules for building component libraries apply to domains. Place domain files in package directories and build them using the `ssc_build` command.

Customizing the Block Name and Appearance

In this section...

“Default Block Display” on page 3-11

“How to Customize the Block Name” on page 3-13

“How to Describe the Block Purpose” on page 3-14

“How to Specify Meaningful Names for the Block Parameters” on page 3-15

“How to Customize the Names and Locations of the Block Ports” on page 3-16

“How to Customize the Block Icon” on page 3-18

“Example — Customized Block Display” on page 3-20

Default Block Display

When you build a custom block, the block name and the parameter names in the block dialog box are derived from the component file elements. The default block icon in the custom library is a rectangle displaying the block name. Ports are based on the nodes, inputs, and outputs defined in the component file.

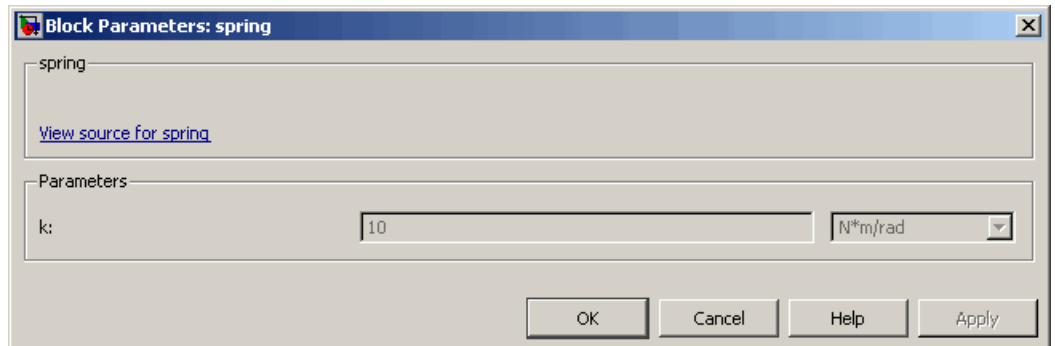
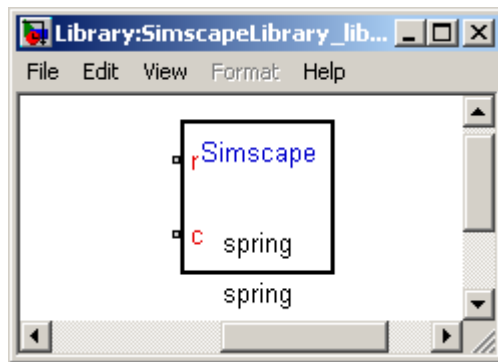
The following example shows a component file, named `spring.ssc`, and the resulting library block and dialog box.

```

component spring
  nodes
    r = foundation.mechanical.rotational.rotational;
    c = foundation.mechanical.rotational.rotational;
  end
  parameters
    k = { 10, 'N*m/rad' };
  end
  variables
    theta = { 0, 'rad' };
    t = { 0, 'N*m' };
    w = { 0, 'rad/s' };
  end
  function setup

```

```
if k < 0
    error( 'Spring rate must be greater than zero' );
end
through( t, r.t, c.t );
across( w, r.w, c.w );
end
equations
t == k * theta;
w == theta.der;
end
end
```



If you click the [View source for spring](#) link, the `spring.ssc` file opens in the MATLAB Editor window.

The following sections show you how to annotate the component file to improve the block cosmetics. You can provide meaningful names for the block itself and its parameters in the dialog box, as well as supply a short description of its purpose. You can also substitute a custom block icon for the default image and change the names and the default orientation of the ports.

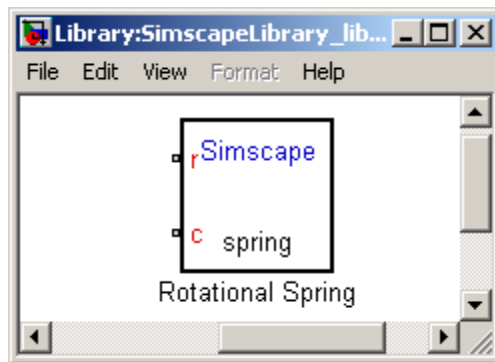
How to Customize the Block Name

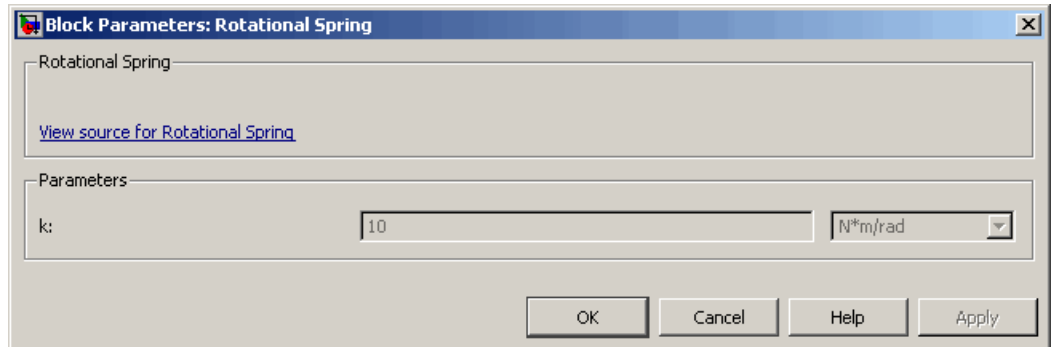
To provide a more descriptive name for the block than the name of the component file, put it on a separate comment line just below the component declaration. The comment line must begin with the % character. The entire content of this line, following the % character, is interpreted as the block name and appears exactly like that in the block icon and at the top of the block dialog box.

For example, if you have the following component file:

```
component spring
  %Rotational Spring
  ...
end
```

these are the resulting block icon and dialog box:





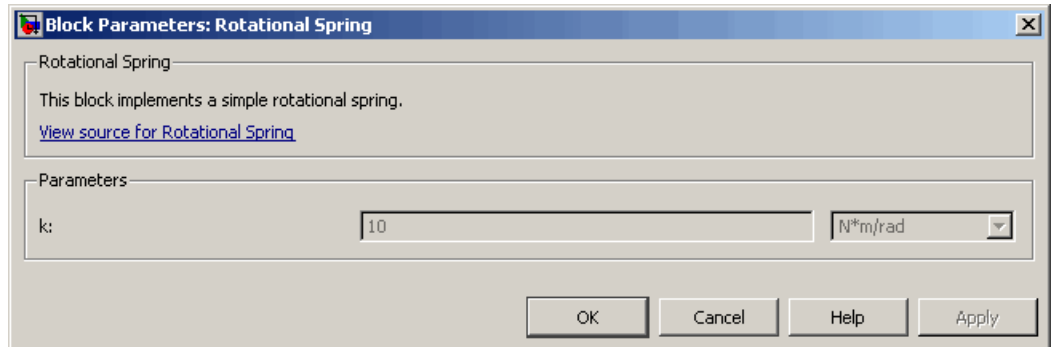
How to Describe the Block Purpose

The previous section describes how the comment line immediately following the component declaration is interpreted as the block name. Any additional comments below that line are interpreted as the block description. You can have more than one line of description comments. Each line must be no longer than 80 characters and must begin with the % character. The entire content of description comments will appear in the block dialog box and in the Library Browser.

For example, if you have the following component file:

```
component spring
%Rotational Spring
% This block implements a simple rotational spring.
...
end
```

this is the resulting block dialog box:



To create a paragraph break in the block description, use a blank commented line:

```
% end of one paragraph
%
% beginning of the next paragraph
```

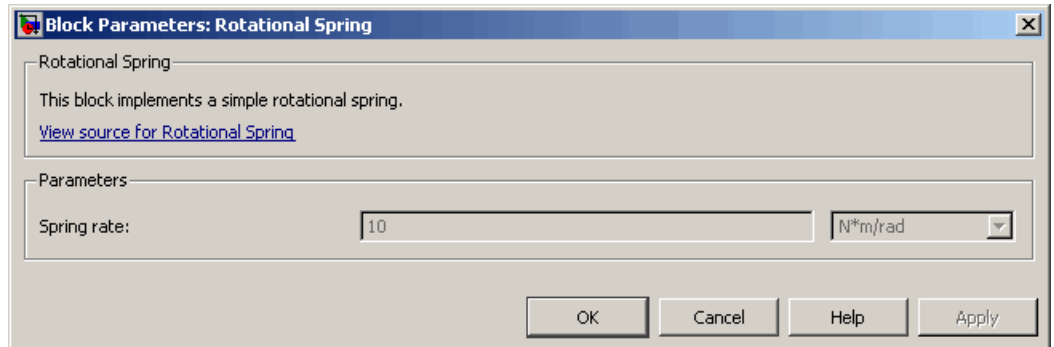
How to Specify Meaningful Names for the Block Parameters

You can specify the name of a block parameter, the way you want it to appear in the block dialog box, as a comment immediately following the parameter declaration. It can be located on the same line or on a separate line. The comment must begin with the % character.

For example, if you have the following component file:

```
component spring
%Rotational Spring
% This block implements a simple rotational spring.
...
parameters
    k = { 10, 'N*m/rad' }; % Spring rate
end
...
end
```

this is the resulting block dialog box:



How to Customize the Names and Locations of the Block Ports

Block ports, both conserving and Physical Signal, are based on the nodes, inputs, and outputs defined in the component file. The default port label corresponds to the name of the node, input, or output, as specified in the declaration block. The default location of all ports is on the left side of the block icon. The ports are spread equidistantly along the block side.



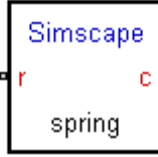

To control the port label and location in the block icon, add a comment immediately following the corresponding node, input, or output declaration. It can be on the same line or on a separate line. The comment must begin with the % character and be of the format `label:location`, where `label` is a string corresponding to the input port name in the block diagram, and `location` is one of the following strings: `left`, `right`, `top`, `bottom`. You can locate all ports either on one side of the block or on two opposite sides, for example `left` and `right`, or `top` and `bottom`. You can omit the location if you want to keep the default location of the port (on the left side).



You can also leave the port label field empty and specify just the location. In this case, the port will not have its name displayed. For example, the following syntax suppresses the port label and locates it on the top of the block icon:

```
r = foundation.mechanical.rotational.rotational; % :top
```

If you specify an empty comment string after a node, input, or output declaration, the corresponding port will not be labelled and will be located on the left side of the block icon.

The following are examples of node declarations and the resulting block icons.

Syntax	Block Icon
<pre> nodes r = foundation.mechanical.rotational.rotational; c = foundation.mechanical.rotational.rotational; end </pre>	 <p>Rotational Spring</p>
<pre> nodes r = foundation.mechanical.rotational.rotational; % rod c = foundation.mechanical.rotational.rotational; % case end </pre>	 <p>Rotational Spring</p>
<pre> nodes r = foundation.mechanical.rotational.rotational; c = foundation.mechanical.rotational.rotational; % c:right end </pre>	 <p>Rotational Spring</p>
<pre> nodes r = foundation.mechanical.rotational.rotational; % rod c = foundation.mechanical.rotational.rotational; % case:r: end </pre>	 <p>Rotational Spring</p>

Syntax	Block Icon
<pre> nodes r = foundation.mechanical.rotational.rotational; % rod c = foundation.mechanical.rotational.rotational; % :right end </pre>	 <p>Rotational Spring</p>
<pre> nodes r = foundation.mechanical.rotational.rotational; % c = foundation.mechanical.rotational.rotational; % case:r: end </pre>	 <p>Rotational Spring</p>

How to Customize the Block Icon

If the subpackage containing the component file (for example, `spring.ssc`) also contains a file named `spring.img`, where *img* is one of the supported image file formats (such as `jpg`, `bmp`, or `png`), then that image file is used for the icon representing this block in the custom library.

The following image file formats are supported:

- `jpg`
- `bmp`
- `png`

If there are multiple image files, the formats take precedence in the order listed above. For example, if the subpackage contains both `spring.jpg` and `spring.bmp`, `spring.jpg` is the image that will appear in the custom library.

Specifying Scaling and Rotation Properties of the Custom Block Icon

When you use an image file to represent a component in the custom block library, the following syntax in the component file lets you specify the scaling and rotation properties of the image file:

```
component name
% [ CustomName [ : scale [ : rotation ] ] ]
...
```

where

<i>name</i>	Component name
<i>CustomName</i>	Customized block name, specified as described in “How to Customize the Block Name” on page 3-13. Leading and trailing white spaces are removed.
<i>scale</i>	A scalar number, for example, 2.0, which specifies the desired scaling of the block icon. When an image file is used as a block icon, by default its shortest size is 40 pixels, with the image aspect ratio preserved. For example, if your custom image is stored in a .jpg file of 80x120 pixels, then the default block icon size will be 40x60 pixels. If you specify a scale of 0.5, then the block icon size will be 20x30 pixels. You cannot specify MATLAB expressions for the scale, just numbers.
<i>rotation</i>	Specifies whether the block icon rotates with the block: <ul style="list-style-type: none"> • rotates means that the icon rotates when you rotate the block. This is the default behavior. • fixed means that the ports rotate when you rotate the block, but the icon always stays in default orientation.

For example, the following syntax

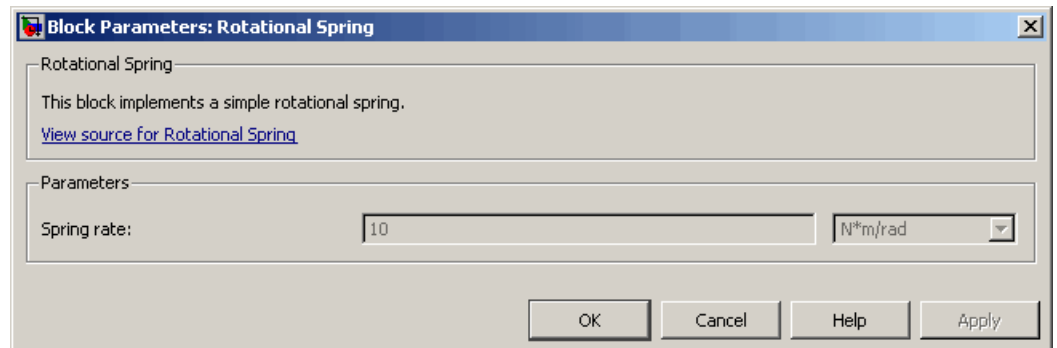
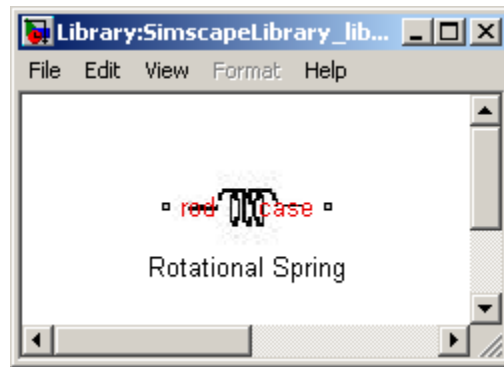
```
component spring
% Rotational Spring : 0.5 : fixed
```

specifies that the spring image size is scaled to half of its default size and always stays in its default orientation, regardless of the block rotation.

Example – Customized Block Display

The following shows a complete example of a component file with annotation and the resulting library block and dialog box. The file is named `spring.ssc`, and the package contains the image file `spring.jpg`, as described in the previous section, “How to Customize the Block Icon” on page 3-18.

```
component spring
% Rotational Spring
% This block implements a simple rotational spring.
nodes
    r = foundation.mechanical.rotational.rotational; % rod
    c = foundation.mechanical.rotational.rotational; % case:right
end
parameters
    k = { 10, 'N*m/rad' }; % Spring rate
end
variables
    theta = { 0, 'rad' };
    t = { 0, 'N*m' };
    w = { 0, 'rad/s' };
end
function setup
    if k < 0
        error( 'Spring rate must be greater than zero' );
    end
    through( t, r.t, c.t );
    across( w, r.w, c.w );
end
equations
    t == k * theta;
    w == theta.der;
end
end
```



Checking File and Model Dependencies

In this section...
“Why Check File and Model Dependencies?” on page 3-22
“Checking Dependencies of Protected Files” on page 3-23
“Checking Simscape File Dependencies” on page 3-23
“Checking Library Dependencies” on page 3-24
“Checking Model Dependencies” on page 3-24

Why Check File and Model Dependencies?

Each Simulink model requires a set of files to run successfully. These files can include referenced models, data files, S-functions, and other files without which the model cannot run. These required files are called *model dependencies*. The Simulink Manifest Tools allow you to analyze a model to determine its model dependencies.

Similarly, Simscape files and custom libraries also depend on certain files to build successfully, or to correctly visualize and execute in MATLAB. These files can include all component files for building a library, domain files, custom image files for blocks or libraries, and so on.

Dependency analysis tools for Simscape files consist of the following command-line options:

- `simscape.dependency.file` — Return the set of existing full path dependency files and missing files for a single Simscape file, for a specific dependency type.
- `simscape.dependency.lib` — Return the set of existing full path dependency files and missing files for a Simscape custom library package. You can optionally specify dependency type and library model file name.
- `simscape.dependency.model` — Return the set of Simscape related dependency files and missing files for a given model containing Simscape and Simulink blocks.

Manifest reports generated using Simulink Manifest Tools also include dependencies for the Simscape blocks present in the model. For more information on the Simulink Manifest Tools, see “Model Dependencies” in the *Simulink User’s Guide*.

Checking Dependencies of Protected Files

If a package contains Simscape protected files, with the corresponding Simscape source files in the same folder, the analysis returns the names of protected files and then analyzes the source files for further dependencies. If the package contains Simscape protected files without the corresponding source files, the protected file names are returned without further analysis.

This way, dependency information is not exposed to a model user, who has only protected files. However, the developer, who has both the source and protected files, is able to perform complete dependency analysis.

Checking Simscape File Dependencies

To check dependencies for a single Simscape file, use the function `simscape.dependency.file`.

For example, consider the following directory structure:

```
- +MySimscapeLibrary
  |-- +MechanicalElements
  |   |-- lib.m
  |   |-- lib.jpg
  |   |-- spring.ssc
  |   |-- spring.jpg
  |   |-- ...
```

The top-level package, `+MySimscapeLibrary`, is located in a directory on the MATLAB path.

To check dependencies for the file `spring.ssc`, type the following at the MATLAB command prompt:

```
[a, b] = simscape.dependency.file('MySimscapeLibrary.MechanicalElements.spring')
```

This command returns two cell arrays of strings: array **a**, containing full path names of existing dependency files (such as `spring.jpg`), and array **b**, containing names of missing files. If none of the files are missing, array **b** is empty.

For more information, see the `simscape.dependency.file` function reference page.

Checking Library Dependencies

To check dependencies for a Simscape library package, use the function `simscape.dependency.lib`.

For example, to return all dependency files for a top-level package `+MySimscapeLibrary`, change your working directory to the folder containing this package and type the following at the MATLAB command prompt:

```
[a, b] = simscape.dependency.lib('MySimscapeLibrary')
```

If you are running this command from a working directory inside the package, you can omit the library name, because it is the only argument, and type:

```
[a, b] = simscape.dependency.lib
```

This command returns two cell arrays of strings: array **a**, containing full path names of all existing dependency files and array **b**, containing names of missing files. If none of the files are missing, array **b** is empty.

To determine which files are necessary to share the library package, type:

```
[a, b] = simscape.dependency.lib('MySimscapeLibrary',Simscape.DependencyType.Simulink)
```

In this case, the arrays **a** and **b** contain all files necessary to build the library, run the models built from its blocks, and visualize them correctly.

Checking Model Dependencies

To perform a complete dependencies check, open the model and from the top menu bar select **Tools > Model Dependencies > Generate Manifest**. The Generate Model Manifest dialog box opens. For more information, see “Model Dependencies” in the *Simulink User’s Guide*.

To check dependencies on Simscape blocks and files only, use the function `simscape.dependency.model`. For example, open the model `dc_motor.mdl` and type:

```
[a b c d] = simscape.dependency.model('dc_motor')
```

This command returns two cell arrays of strings and two lists of structures. Array `a` contains full path names of all existing dependency files. Array `b` contains names of missing files. Structure lists `c` and `d` indicate reference types for existing and missing reference files, respectively. Each structure includes a field `'names'` as a list of file names causing the reference, and a field `'type'` as the reference type for each file. Two reference types are used: `'Simscape component'` indicates reference from a model block. `'Simscape'` indicates reference from a file.

If none of the files are missing, array `b` and list `d` are empty.

Case Study – Creating a Basic Custom Block Library

In this section...
“Getting Started” on page 3-26
“Building the Custom Library” on page 3-27
“Adding a Block” on page 3-27
“Adding Detail to a Component” on page 3-28
“Adding a Component with an Internal Variable” on page 3-30
“Customizing the Block Icon” on page 3-32

Getting Started

This case study explains how to build your own library of custom blocks based on component files. It uses a demo library of capacitor models. The library makes use of the Simscape Foundation electrical domain, and defines three simple components. For more advanced topics, including adding multiple levels of hierarchy, adding new domains, and customizing the appearance of a library, see “Case Study — Creating an Electrochemical Library” on page 3-33.

The demo library comes built and on your path so that it is readily executable. However, it is recommended that you copy the source files to a new directory, for which you have write permission, and add that directory to your MATLAB path. This will allow you to make changes and rebuild the library for yourself. The source files for the demo library are in the following package directory:

```
matlabroot/toolbox/physmod/simscape/simscapedemos/+Capacitors
```

where *matlabroot* is the MATLAB root directory on your machine, as returned by entering

```
matlabroot
```

in the MATLAB Command Window.

After copying the files, change the directory name `+Capacitors` to another name, for example `+MyCapacitors`, so that your copy of the library builds with a unique name.

Building the Custom Library

To build the library, type

```
ssc_build MyCapacitors
```

in the MATLAB Command Window. If building from within the +MyCapacitors package directory, you can omit the argument and type just

```
ssc_build
```

When the build completes, open the generated library by typing

```
MyCapacitors_lib
```

For more information on the library build process, see “Adding Custom Block Libraries Generated from Simscape Component Files” on page 3-2.

Adding a Block

To add a block, write a corresponding component file and place it in the package directory. For example, the Ideal Capacitor block in your MyCapacitors_lib.mdl is produced by the IdealCapacitor.ssc file. Open this file in the MATLAB Editor and examine its contents.

```
component IdealCapacitor
% Ideal Capacitor
% Models an ideal (lossless) capacitor. The output current I is related
% to the input voltage V by  $I = C \cdot dV/dt$  where C is the capacitance.

% Copyright 2008 The MathWorks, Inc.

nodes
    p = foundation.electrical.electrical; % +:top
    n = foundation.electrical.electrical; % -:bottom
end
parameters
    C = { 1, 'F' }; % Capacitance
    V0 = { 0, 'V' }; % Initial voltage
end
variables
```

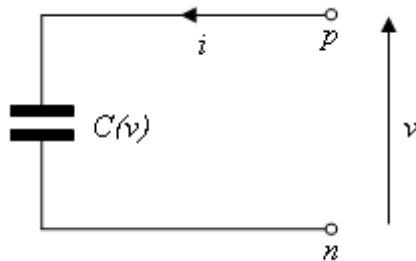
```
i = { 0, 'A' }; % Current through variable
v = { 0, 'V' }; % Voltage across variable
end
function setup
    if C <= 0
        error( 'Capacitance must be greater than zero' )
    end
    through( i, p.i, n.i ); % Through variable i from node p to node n
    across( v, p.v, n.v ); % Across variable v from p to n
    v = V0;
end
equations
    i == C*v.der; % Equation
end
end
```

First, let us examine the elements of the component file that affect the block appearance. Double-click the Ideal Capacitor block in the `MyCapacitors_lib.mdl` to open its dialog box, and compare the block icon and dialog box to the contents of the `IdealCapacitor.ssc` file. The block name, Ideal Capacitor, is taken from the comment on line 2. The comments on lines 3 and 4 are then taken to populate the block description in the dialog box. The block ports are defined by the nodes section. The comment expressions at the end of each line control the port label and location. Similarly in the parameters section, the comments are used to define parameter names in the block dialog box. For details, see “Customizing the Block Name and Appearance” on page 3-11.

Also notice that in the setup section there is a check to ensure that the capacitance value is always greater than zero. This is good practice to ensure that a component is not used outside of its domain of validity. The Simscape Foundation library blocks have such checks implemented where appropriate.

Adding Detail to a Component

In this demo library there are two additional components that can be used for ultracapacitor modeling. These components are evolutions of the Ideal Capacitor. It is good practice to incrementally build component models, adding and testing additional features as they are added.



Ideal Ultracapacitor

Ultracapacitors, as their name suggests, are capacitors with a very high capacitance value. The relationship between voltage and charge is not constant, unlike for an ideal capacitor. Suppose a manufacturer data sheet gives a graph of capacitance as a function of voltage, and that capacitance increases approximately linearly with voltage from the 1 farad at zero volts to 1.5 farads when the voltage is 2.5 volts. If the capacitance voltage is denoted v , then the capacitance can be approximated as:

$$C = 1 + 0.2 \cdot v$$

For a capacitor, current i and voltage v are related by the standard equation

$$i = C \frac{dv}{dt}$$

and hence

$$i = (C_0 + C_v \cdot v) \frac{dv}{dt}$$

where $C_0 = 1$ and $C_v = 0.2$. This equation is implemented by the following line in the equation section of the Simscape file `IdealUltraCapacitor.ssc`:

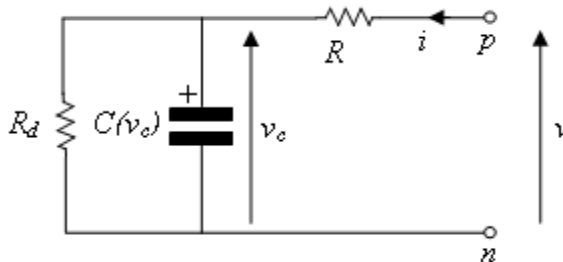
```
i == (C0 + Cv*v)*v.der;
```

In order for the Simscape software to interpret this equation, the variables (v and i) and the parameters (C_0 and C_v) must be defined in the declaration

section. For more information, see “Declaring Component Variables” on page 2-7 and “Declaring Component Parameters” on page 2-8.

Adding a Component with an Internal Variable

Implementing some component equations requires the use of internal variables. An example is when implementing an ultracapacitor with resistive losses. There are two resistive terms, the effective series resistance R , and the self-discharge resistance R_d . Because of the topology, it is not possible to directly express the capacitor equations in terms of the through and across variables i and v .



Ultracapacitor with Resistive Losses

This block is implemented by the component file `LossyUltraCapacitor.ssc`. Open this file in the MATLAB Editor and examine its contents.

```

component LossyUltraCapacitor
% Lossy Ultracapacitor
% Models an ultracapacitor with resistive losses. The capacitance C
% depends on the voltage V according to  $C = C_0 + V \cdot dC/dV$ . A
% self-discharge resistance is included in parallel with the capacitor,
% and an equivalent series resistance in series with the capacitor.

% Copyright 2008 The MathWorks, Inc.

nodes
    p = foundation.electrical.electrical; % +:top
    n = foundation.electrical.electrical; % -:bottom
end
    
```



```

parameters
    C0 = { 1, 'F' }; % Nominal capacitance C0 at V=0
    Cv = { 0.2, 'F/V' }; % Rate of change of C with voltage V
    R = {2, 'Ohm' }; % Effective series resistance
    Rd = {500, 'Ohm' }; % Self-discharge resistance
    V0 = { 0, 'V' }; % Initial voltage
end
variables
    i = { 0, 'A' }; % Current through variable
    v = { 0, 'V' }; % Voltage across variable
    vc = { 0, 'V' }; % Internal variable for capacitor voltage
end
function setup
    if C0 <= 0
        error( 'Nominal capacitance C0 must be greater than zero' )
    end
    if R <= 0
        error( 'Effective series resistance must be greater than zero' )
    end
    if Rd <= 0
        error( 'Self-discharge resistance must be greater than zero' )
    end
    through( i, p.i, n.i ); % Through variable i from node p to node n
    across( v, p.v, n.v ); % Across variable v from p to n
    vc = V0;
end
equations
    i == (C0 + Cv*v)*vc.der + vc/Rd; % Equation 1
    v == vc + i*R; % Equation 2
end
end

```

The additional variable is used to denote the voltage across the capacitor, v_c . The equations can then be expressed in terms of v , i , and v_c using Kirchhoff's current and voltage laws. Summing currents at the capacitor + node gives the first Simscape equation:

$$i == (C0 + Cv*v)*v.der + vc/Rd;$$

Summing voltages gives the second Simscape equation:

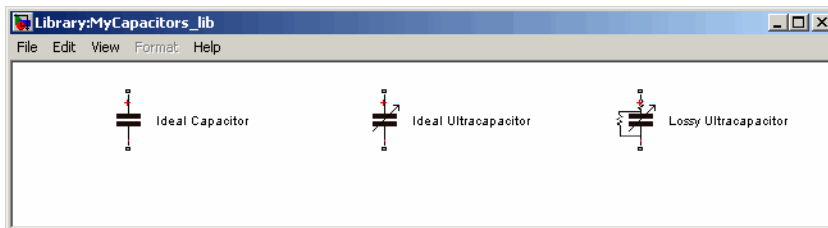
```
v == vc + i*R;
```

As a check, the number of equations required for a component used in a single connected network is given by the sum of the number of ports plus the number of internal variables minus one. This is not necessarily true for all components (for example, one exception is mass), but in general it is a good rule of thumb. Here this gives $2 + 1 - 1 = 2$.

In the Simscape file, the initial condition (initial voltage in this example) is applied to variable vc and not v . This is because initial conditions should be applied only to differential variables. In this case, vc is readily identifiable as the differential variable as it has the `der` (differentiator) operator applied to it.

Customizing the Block Icon

The capacitor blocks in the demo library `MyCapacitors_lib.mdl` have icons associated with them.



During the library build, if there is an image file in the directory with the same name as the Simscape component file, then this is used to define the icon for the block. For example, the Ideal Capacitor block defined by `IdealCapacitor.ssc` uses the `IdealCapacitor.jpg` to define its block icon. If you do not include an image file, then the block displays its name in place of an icon. For details, see “How to Customize the Block Icon” on page 3-18.

Case Study — Creating an Electrochemical Library

In this section...

“Getting Started” on page 3-33

“Building the Custom Library” on page 3-34

“Defining a New Domain” on page 3-34

“Structuring the Library” on page 3-37

“Defining a Reference Component” on page 3-37

“Defining an Ideal Source Component” on page 3-38

“Defining Measurement Components” on page 3-39

“Defining Basic Components” on page 3-41

“Defining a Cross-Domain Interfacing Component” on page 3-44

“Customizing the Appearance of the Library” on page 3-46

“Using the Custom Components to Build a Model” on page 3-47

“References” on page 3-47

Getting Started

This case study explores more advanced topics of building custom Simscape libraries. It uses a demo library for modeling electrochemical systems. The library introduces a new electrochemical domain and defines all of the fundamental components required to build electrochemical models, including an electrochemical reference, through and across sensors, sources, and a cross-domain component. The example illustrates some of the salient features of Physical Networks modeling, such as selection of Through and Across variables and how power is converted between domains. We suggest that you work through the previous section, “Case Study — Creating a Basic Custom Block Library” on page 3-26, before looking at this more advanced example.

The demo library comes built and on your path so that it is readily executable. However, it is recommended that you copy the source files to a new directory, for which you have write permission, and add that directory to your MATLAB path. This will allow you to make changes and rebuild the library for yourself. The source files for the demo library are in the following package directory:

```
matlabroot/toolbox/physmod/simscape/simscapedemos/+ElectroChem
```

where *matlabroot* is the MATLAB root directory on your machine, as returned by entering

```
matlabroot
```

in the MATLAB Command Window.

After copying the files, change the directory name `+ElectroChem` to another name, for example `+MyElectroChem`, so that your copy of the library builds with a unique name.

Building the Custom Library

To build the library, type

```
ssc_build MyElectroChem
```

in the MATLAB Command Window. If building from within the `+MyElectroChem` package directory, you can omit the argument and type just

```
ssc_build
```

When the build completes, open the generated library by typing

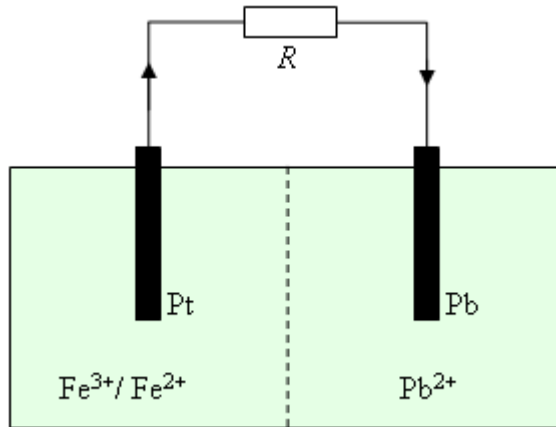
```
MyElectroChem_lib
```

For more information on the library build process, see “Adding Custom Block Libraries Generated from Simscape Component Files” on page 3-2.

Defining a New Domain

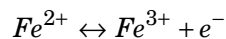
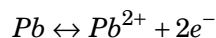
Simscape software comes with several Foundation domains, such as mechanical translational, mechanical rotational, electrical, hydraulic, and so on. Where possible, use these predefined domains. For example, when creating new electrical components, use the Foundation electrical domain `foundation.electrical.electrical`. This ensures that your components can be connected to the standard Simscape blocks.

As an example of an application requiring the addition of a new domain, consider a battery where the underlying equations involve both electrical and chemical processes [1].

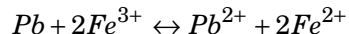


Electrochemical Battery Driving a Resistive Load R

Two half-cells are separated by a membrane that prevents the ions flowing between cells, and hence electrons flow from the solid lead anode to the platinum cathode. The two half-cell reactions are:



The current results in the lead being oxidized and the iron being reduced, with the overall reaction given by:



The chemical reaction can be modeled using the network concepts of Through and Across variables (for details, see “Basic Principles of Modeling Physical Networks” in the *Simscape User’s Guide*). The Through variable represents flow, and the Across variable represents effort. When selecting the Through

and Across variables, you should use SI units and the product of the two variables is usually chosen to have units of power.

In the electrochemical reactions, an obvious choice for the Through variable is the molar flow rate \dot{n} of ions, measured in SI units of mol/s. The corresponding Across variable is called chemical potential, and must have units of J/mol to ensure that the product of Through and Across variables has units of power, J/s. The chemical potential or Gibb's free energy per mol is given by:

$$\mu = \mu_0 + RT \ln a$$

where μ_0 is the standard state chemical potential, R is the perfect gas constant, T is the temperature, and a is the activity. In general, the activity can be a function of a number of different parameters, including concentration, temperature, and pressure. Here it is assumed that the activity is proportional to the molar concentration defined as number of moles of solute divided by the mass of solvent.

To see the electrochemical domain definition, open the Simscape file +MyElectroChem/ElectroChem.ssc.

```
domain ElectroChem
% Define through and across variables for the electrochemical domain

% Copyright 2008 The MathWorks, Inc.

variables
    % Chemical potential
    mu = { 1.0 'J/mol' };
end

variables(Balancing = true)
    % Molar flow
    ndot = { 1.0 'mol/s' };
end

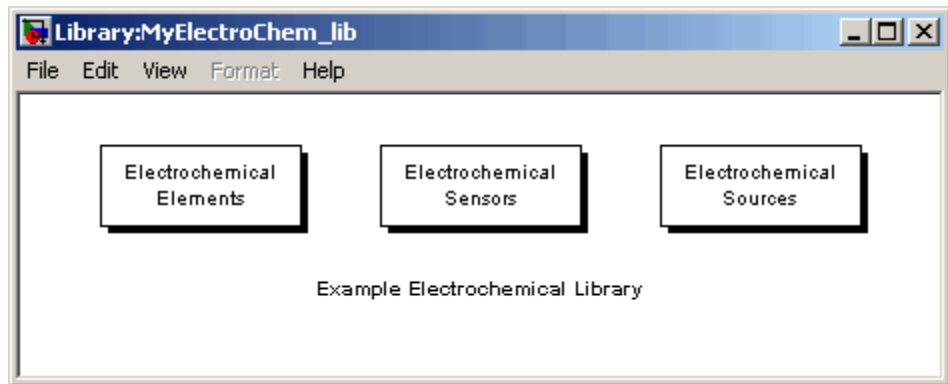
end
```

The molar fundamental dimension and unit is predefined in the Simscape unit registry. If it had not been, then you could have added it with:

```
pm_adddimension('mole','mol')
```

Structuring the Library

It is good practice to structure a library by adding hierarchy. To do this, you can subdivide the package directory into subdirectories, each subdirectory name starting with the + character. If you look at the +MyElectroChem directory, you will see that it has subdirectories +Elements, +Sensors, and +Sources. Open the library by typing MyElectroChem_lib, and you will see the three corresponding sublibraries.



Defining a Reference Component

A physical network must have a reference block, against which Across variables are measured. So, for example, the Foundation library contains the Electrical Reference block for the electrical domain, Mechanical Rotational Reference block for the rotational mechanical domain, and so on. The electrochemical zero chemical potential is defined by the component file +MyElectroChem/+Elements/Reference.ssc.

```
component Reference
% Chemical Reference
% Port A is a zero chemical potential reference port.

% Copyright 2008 The MathWorks, Inc.
```

```
nodes
    A = ElectroChem.ElectroChem; % A:top
end
variables
    ndot = { 1.0 'mol/s' };
end
function setup
    through( ndot, A.ndot, [] );
end
equations
    % Equations
    A.mu == 0;
end
end
```

The component has one electrochemical port, named A. The chemical potential is defined as zero by equation:

```
mu == 0;
```

Variable mu is defined as the across variable from the A port to zero with the following line at setup, the empty square brackets denoting the zero reference:

```
across( mu, A.mu, [] );
```

Defining an Ideal Source Component

An ideal Across source provides a constant value for the Across variable regardless of the value of the Through variable. In the electrical domain, this corresponds to the DC Voltage Source block in the Foundation library. In the demo library, the component file +MyElectroChem/+Sources/ChemPotentialSource.ssc implements the equivalent source for the chemical domain.

```
component ChemPotentialSource
% Constant Potential Source
% Provides a constant chemical potential between ports A and B.

% Copyright 2008 The MathWorks, Inc.

nodes
```



```

    A = ElectroChem.ElectroChem; % A:top
    B = ElectroChem.ElectroChem; % B:bottom
end
parameters
    mu0 = {0, 'J/mol'}; % Chemical potential
end
variables
    ndot = { 0, 'mol/s' };
    mu = { 0, 'J/mol' };
end
function setup
    across( mu, A.mu, B.mu ); % Assign mu to across variable from A to B
    through( ndot, A.ndot, B.ndot ); % Assign ndot to through variable from A to B
end
equations
    % Equations
    mu == mu0;
end

end

```

The dual of an ideal Across source is an ideal Through source, which maintains the Through variable to some set value regardless of the value of the Across variable. In the electrical domain, this corresponds to the DC Current Source block in the Foundation library. In the demo library, this source is not implemented.

Defining Measurement Components

Every domain requires both a Through and an Across measurement block. In the demo library, the component file `+MyElectroChem/+Sensors/SensorThrough.ssc` implements a molar flow rate sensor.

```

component SensorThrough
% Molar Flow Sensor
% Returns the value of the molar flow between the A and the B port
% to the physical signal port PS.

% Copyright 2008 The MathWorks, Inc.

```

```
nodes
    A = ElectroChem.ElectroChem; % A:top
    B = ElectroChem.ElectroChem; % B:bottom
end
outputs
    out = { 0, 'mol/s' }; % PS:top
end
variables
    ndot = { 0, 'mol/s' };
    mu = { 0, 'J/mol' };
end
function setup
    through( ndot, A.ndot, B.ndot ); % Assign ndot to through variable from A to B
    across( mu, A.mu, B.mu );      % Assign mu to across variable from A to B
end
equations
    % Equations
    mu == 0;      % No potential drop
    out == ndot; % Equate value of molar flow to PS output
end

end
```

The flow rate is presented as a Physical Signal, which can then in turn be passed to Simulink via a PS-Simulink Converter block. The equation section requires two equations—one to assign the value of the Through variable to the Physical Signal output, and one to define the relationship between Through and Across variables for the sensor. In this case, an ideal flow sensor has zero potential drop, that is $\mu == 0$, where μ is the chemical potential.

The component file `+MyElectroChem/+Sensors/SensorAcross.ssc` implements a chemical potential sensor.

```
component SensorAcross
% Chemical Potential Sensor
% Returns the value of the chemical potential across the A and B ports
% to the physical signal port PS.

% Copyright 2008 The MathWorks, Inc.
```

```

nodes
    A = ElectroChem.ElectroChem; % A:top
    B = ElectroChem.ElectroChem; % B:bottom
end
outputs
    out = { 0, 'J/mol' }; % PS:top
end
variables
    ndot = { 0, 'mol/s' };
    mu = { 0, 'J/mol' };
end
function setup
    through( ndot, A.ndot, B.ndot ); % Assign ndot to through variable from A to B
    across( mu, A.mu, B.mu );      % Assign mu to across variable from A to B
end
equations
    % Equations
    ndot == 0; % Draws no molar flow
    out == mu; % Equate value of chemical potential difference to PS output
end

end

```

The chemical potential is presented as a Physical Signal, which can then in turn be passed to Simulink via a PS-Simulink Converter block. The equation section requires two equations—one to assign the value of the Across variable to the Physical Signal output, and one to define the relationship between Through and Across variables for the sensor. In this case, an ideal chemical potential sensor draws no flow, that is $\text{ndot} == 0$, where ndot is the flow rate.

Defining Basic Components

Having created the measurement and reference blocks, the next step is to create blocks that define behavioral relationships between the Through and Across variables. In the electrical domain, for example, such components are resistor, capacitor, and inductor.

As an example of a basic electrochemical component, consider the chemical reduction or oxidation of an ion, which can be thought of as the electrochemical

equivalent of a nonlinear capacitor. The defining equations in terms of Through and Across variables v and μ are:

$$\dot{n} = v$$

$$a = \frac{n}{C_0 M}$$

$$\mu = \mu_0 + RT \ln a$$

where n is the number of moles of the ion, C_0 is the standard concentration of 1 mol/kg, and M is the mass of the solute.

To see the implementation of these equations, open the file `+MyElectroChem/+Elements/ChemEnergyStore.ssc`.

```

component ChemEnergyStore
% Chemical Energy Store
% Represents a solution of dissolved ions. The port A presents the
% chemical potential defined by  $\mu_0 + \log(n/(C_0*M))*R*T$  where  $\mu_0$  is the
% standard state oxidising potential,  $n$  is the number of moles of the ion,
%  $C_0$  is the standard concentration of 1 mol/kg,  $M$  is the mass of solvent,
%  $R$  is the universal gas constant, and  $T$  is the temperature.

% Copyright 2008 The MathWorks, Inc.

nodes
    A = ElectroChem.ElectroChem; % A:top
end
parameters
    mu0 = {-7.42e+04, 'J/mol'}; % Standard state oxidising potential
    n0 = {0.01, 'mol'}; % Initial quantity of ions
    m_solvent = {1, 'kg'}; % Mass of solvent
    T = {300, 'K'}; % Temperature
end
parameters (Access=private)
    R = {8.314472, '(J/K)/mol'}; % Universal gas constant
    C0 = {1, 'mol/kg'}; % Standard concentration
    n1 = {1e-10, 'mol'}; % Minimum number of moles

```

```

end
variables
    ndot = { 0, 'mol/s' };
    mu = { 0, 'J/mol' };
    n = {0.01, 'mol'}; % Quantity of ions
end
function setup
    through( ndot, A.ndot, [] ); % Through variable ndot
    across( mu, A.mu, [] ); % Across variable mu
    n = n0;
end
equations
    % Equations
    n.der == ndot;
    if n > n1
        mu == mu0 + log(n/(C0*m_solvent))*R*T;
    else
        mu == mu0 + (log(n1/(C0*m_solvent)) + n/n1 - 1)*R*T;
    end
end
end
end

```

This component introduces two Simscape language features not yet used in the blocks looked at so far. These are:

- Use of a conditional statement in the equation section. This is required to prevent taking the logarithm of zero. Hence if the molar concentration is less than the specified level n_1 , then the operand of the logarithm function is limited. Without this protection, the solver could perturb the value of n to zero or less.
- Definition of private parameters that can be used in the setup or equation sections. Here the Universal Gas constant (R) and the Standard Concentration (C_0) are defined as private parameters. Their values could equally well be used directly in the equations, but this would reduce readability of the definition. Similarly, the lower limit on the molar concentration n_1 is also defined as a private parameter, but could equally well have been exposed to the user.

Defining a Cross-Domain Interfacing Component

Cross-domain blocks allow the interchange of energy between domains. For example, the Rotational Electromechanical Converter block in the Foundation library converts between electrical and rotational mechanical energy. To relate the two sets of Through and Across variables, two equations are required. The first comes from an underlying physical law, and the second from summing the powers from the two domains into the converter, which must total zero.

As an example of an interfacing component, consider the electrochemical half-cell. The chemical molar flow rate and the electrical current are related by Faraday's law, which requires that:

$$v = \frac{i}{zF}$$

where v is the molar flow rate, i is the current, z is the number of electrons per ion, and F is the Faraday constant. The second equation comes from equating the electrical and chemical powers:

$$(V_2 - V_1)i = (\mu_2 - \mu_1)v$$

which can be rewritten as:

$$(V_2 - V_1) = (\mu_2 - \mu_1) \frac{v}{i} = \frac{\mu_2 - \mu_1}{zF}$$

This is the Nernst equation written in terms of chemical potential difference, $(\mu_2 - \mu_1)$. These chemical-electrical converter equations are implemented by the component file +MyElectroChem/+Elements/Chem2Elec.ssc.

```

component Chem2Elec
% Chemical to Electrical Converter
% Converts chemical energy into electrical energy (and vice-versa
% assuming no losses. The electrical current flow i is related to the
% molar flow of electrons ndot by i = -ndot*z*F where F is the Faraday
% constant and z is the number of exchanged electrons.

% Copyright 2008-2009 The MathWorks, Inc.
```

```

nodes
    p = foundation.electrical.electrical; % +:top
    n = foundation.electrical.electrical; % -:top
    A = ElectroChem.ElectroChem; % A:bottom
    B = ElectroChem.ElectroChem; % B:bottom
end
parameters
    z = {1, '1'}; % Number of exchanged electrons
end
parameters (Access=private)
    F = {9.6485309e4, 'c/mol'}; % Faraday constant
end
variables
    i = { 0, 'A' };
    v = { 0, 'V' };
    ndot = { 0, 'mol/s' };
    mu = { 0, 'J/mol' };
end
function setup
    through( i, p.i, n.i ); % Through variable i from node p to node n
    across( v, p.v, n.v ); % Across variable v from p to n
    through( ndot, A.ndot, B.ndot ); % Through variable ndot from node A to node B
    across( mu, A.mu, B.mu ); % Across variable mu from A to B
end
equations
    % Equations
    let
        k = 1/(z*F);
    in
        v == k*mu; % From equating power
        ndot == -k*i; % Balance electrons (Faraday's Law)
    end
end
end

```

Note the use of the `let-in-end` construction in the component equations. An intermediate term `k` is declared as

$$k = \frac{1}{zF}$$

It is then used in both equations in the expression clause that follows.

This component has four ports but only two equations. This is because the component interfaces two different physical networks. Each of the networks has two ports and one equation, thus satisfying the requirement for $n-1$ equations, where n is the number of ports. In the case of a cross-domain component, the two equations are coupled, thereby defining the interaction between the two physical domains.

The Faraday constant is a hidden parameter, because it is a physical constant that block users would not need to change. Therefore, it will not appear in the block dialog box generated from the component file.

Customizing the Appearance of the Library

The library can be customized using `lib.m` files. A `lib.m` file located in the top-level package directory can be used to add annotations. The name of the top-level library model is constructed automatically during the build process based on the top-level package name, as `package_lib.mdl`, but you can add a more descriptive name to the top-level library as an annotation. For example, open `+MyElectroChem/lib.m` in the MATLAB Editor. The following line annotates the top-level library with its name:

```
libInfo.Annotation = sprintf('Example Electrochemical Library')
```

In the electrochemical library example, `lib.m` files are also placed in each subpackage directory to customize the name and appearance of respective sublibraries. For example, open `+MyElectroChem/+Sensors/lib.m` in the MATLAB Editor. The following line causes the sublibrary to be named `Electrochemical Sensors`:

```
libInfo.Name = 'Electrochemical Sensors';
```

In the absence of the `lib.m` file, the library would be named after the subpackage name, that is, `Sensors`. For more information, see “Customizing the Library Name and Appearance” on page 3-5.

Using the Custom Components to Build a Model

The Model Using a Customized Electrochemical Library demo (`ssc_electrochemical_battery`) uses the electrochemical library to model a lead-iron battery. See the demo help for further information.

References

- [1] Pêcheux, F., B. Allard, C. Lallement, A. Vachoux, and H. Morel. “Modeling and Simulation of Multi-Discipline Systems using Bond Graphs and VHDL-AMS.” International Conference on Bond Graph Modeling and Simulation (ICBGM). New Orleans, USA, 23–27 Jan. 2005.

S

- Simscape™ language 1-2
 - creating custom block libraries 3-2
 - creating sublibraries 3-5
 - file structure 1-8
- syntax 1-10
- turning component files into Simscape™
 - blocks 3-2
- workflows 1-3