Simscape™
Language Guide

R2013a
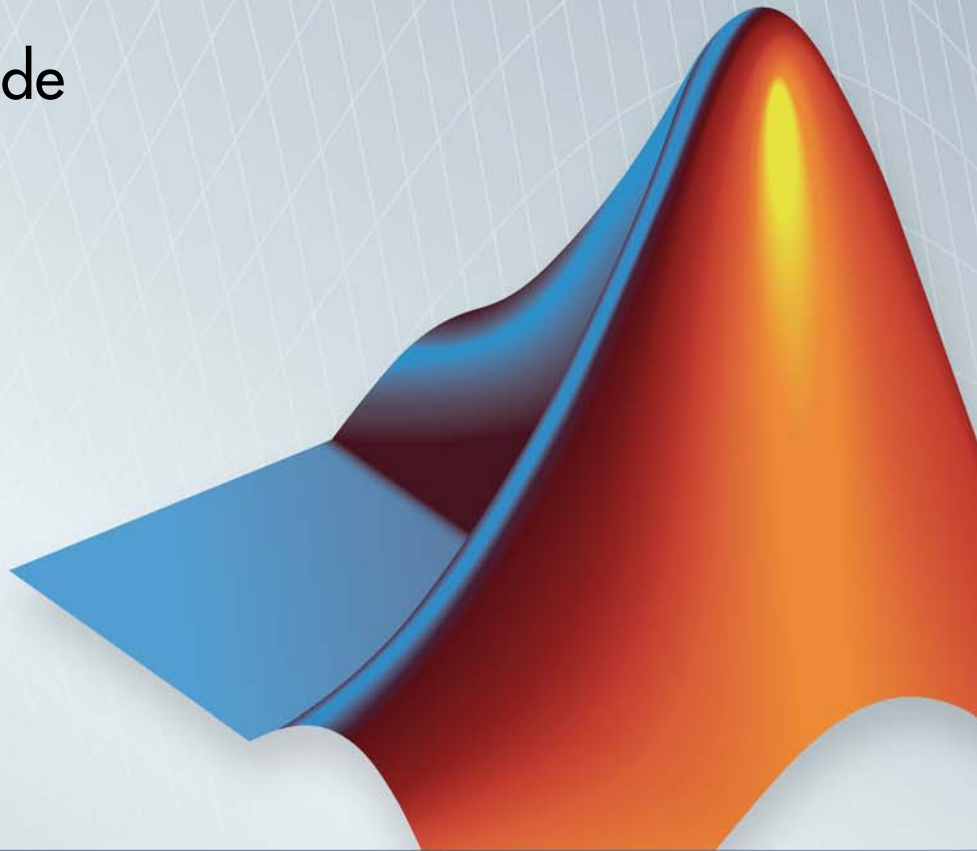
MATLAB&SIMULINK®

MathWorks®

How to Contact 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

Trademarks

Patents

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)
September 2010	Online only	Revised for Version 3.4 (Release 2010b)
April 2011	Online only	Revised for Version 3.5 (Release 2011a)
September 2011	Online only	Revised for Version 3.6 (Release 2011b)
March 2012	Online only	Revised for Version 3.7 (Release 2012a)
September 2012	Online only	Revised for Version 3.8 (Release 2012b)
March 2013	Online only	Revised for Version 3.9 (Release 2013a)

Contents

Simscape Language Fundamentals

1

Creating Custom Components and Domains

2

Simscape File Deployment

3

Language Reference

4

Simscape Foundation Domains

5

Index

Simscape Language Fundamentals

- "What Is the Simscape Language?" on page 1-2
- "Typical Simscape Language Tasks" on page 1-6
- "About Simscape Files" on page 1-8
- "Creating a New Physical Domain" on page 1-13
- "Creating Custom Components" on page 1-15

What Is the Simscape Language?

Overview

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.

Model Linear Resistor in Simscape Language

Let us discuss how modeling in Simscape language works, using a linear resistor as an example.

A linear resistor is a simple electrical component, described by the following equation:

$$V = I \cdot R$$

where

V	Voltage across the resistor
I	Current through the resistor
R	Resistance

A Simscape file that implements such a linear resistor might look as follows:

```
component my_resistor
% Linear Resistor
% The voltage-current (V-I) relationship for a linear resistor is V=I*R,
% where R is the constant resistance in ohms.
%
% The positive and negative terminals of the resistor are denoted by the
% + and - signs respectively.

  nodes
    p = foundation.electrical.electrical; % +:left
    n = foundation.electrical.electrical; % -:right
  end
  variables
    i = { 0, 'A' };
    v = { 0, 'V' };
  end
  parameters
    R = { 1, 'Ohm' };   % Resistance
  end

  function setup
    across( v, p.v, n.v );
    through( i, p.i, n.i );
    if R <= 0
        error( 'Resistance value must be greater than zero' );
    end
  end

  equations
    v == i*R;
  end

end
```

Let us examine the structure of the Simscape file `my_resistor.ssc`.

The first line indicates that this is a component file, and the component name is `my_resistor`.

Following this line, there are optional comments that customize the block name and provide a short description in the block dialog box.

The next section of the Simscape file is the declaration section. For the linear resistor, it declares:

- Two electrical nodes, `p` and `n` (for + and – terminals, respectively)

- Through and Across variables, current `i` and voltage `v`, to be connected to the electrical domain at setup

- Parameter `R`, with a default value of `1 Ohm`, specifying the resistance value. This parameter appears in the dialog box of the block generated from the component file, and can be modified when building and simulating a model. The comment immediately following the parameter declaration, `Resistance`, specifies how the name of the block parameter appears in the dialog box.

The following section is setup. In this case, it serves the two purposes:

- Establishes relationships between the component variables and nodes (and therefore domain variables) by using `across` and `through` functions.

- Performs parameter validation, by checking that the resistance value is greater than zero.

The final section contains the equation `v == i*R`. It defines the mathematical relationship between the component Through and Across variables, current `i` and voltage `v`, and the parameter `R`. The `==` operand used in the equation specifies continuous mathematical equality between the left- and right-hand side expressions. This means that the equation does not represent assignment but rather a symmetric mathematical relationship between the left- and right-hand operands. This equation is evaluated continuously throughout the simulation.

The following illustration shows the resulting custom block, generated from this component file.

Linear Resistor
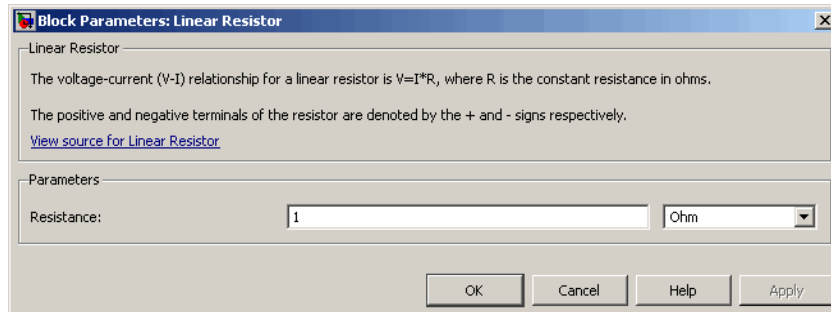
To learn more about writing Simscape files and converting your textual components into libraries of additional Simscape blocks, refer to the following table.

For...	See...
Declaration semantics, rules, and examples	"Declaring Domains and Components" on page 2-2
Purpose and examples of the setup section	"Defining Component Setup" on page 2-16
Detailed information on writing component equations	"Defining Component Equations" on page 2-21
Annotating the component file to improve the generated block cosmetics and usability	"Customizing the Block Name and Appearance" on page 3-11
Generating Simscape blocks from component files	"Generate Custom Block Libraries from Simscape Component Files" on page 3-2

Typical Simscape Language 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.

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	"Creating Custom Components" on page 1-15	"Declare a Spring Component" on page 2-14
	"Declaring Domains and Components" on page 2-2	"Mechanical Component — Spring" on page 2-56
	"Defining Component Setup" on page 2-16	"Electrical Component — Ideal Capacitor" on page 2-57
	"Defining Component Equations" on page 2-21	"No-Flow Component — Voltage Sensor" on page 2-59
		"Grounding Component — Electrical Reference" on page 2-60
Create a custom component model constructed of other components	"Creating Composite Components" on page 2-47	"Composite Component — DC Motor" on page 2-61
	"Declaring Member Components" on page 2-47	
	"Parameterizing Composite Components" on page 2-48	
	"Specifying Component Connections" on page 2-51	

Task	Background Information	Examples
Add a custom block library to Simscape libraries	"Generate Custom Block Libraries from Simscape Component Files" on page 3-2 "Using Source Protection for Simscape Files" on page 3-3 "Customizing the Block Name and Appearance" on page 3-11	"Create a Custom Block Library" on page 3-8 "Custom Block Display" on page 3-20
Define a new domain, with associated Through and Across variables, and then use it in custom components	"Creating a New Physical Domain" on page 1-13 "Declaring Domains and Components" on page 2-2	"Declare a Mechanical Rotational Domain" on page 2-13 "Propagation of Domain Parameters" on page 2-67
Create a component that supplies domain-wide parameters (such as fluid temperature) to the rest of the model	"Working with Domain Parameters" on page 2-67	"Source Components" on page 2-68

About Simscape Files

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. 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 component 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.

For example, to implement a variable hydraulic orifice that is different from the one in the Simscape Foundation library, you can create a component model, `MyVarOrifice.ssc`, based on the standard hydraulic domain included

in the Foundation library. However, to implement a thermohydraulic orifice, you need to create a domain model first, `thermohydraulic.ssc` (a custom hydraulic domain that accounts for fluid temperature), and then create the component model that references it, `MyThhOrifice.ssc`, as well as all the other component models based on this custom domain and needed for modeling thermohydraulic systems.

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
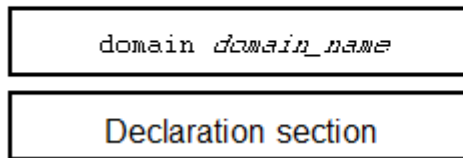
Component file structure

domain *domain_name*
Declaration section

component *component_name*
Declaration section
Setup section
Structure section
Equation section
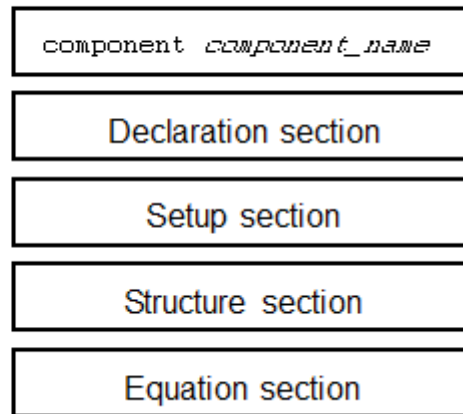
A Simscape file splits the model description into the following pieces:

- *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.
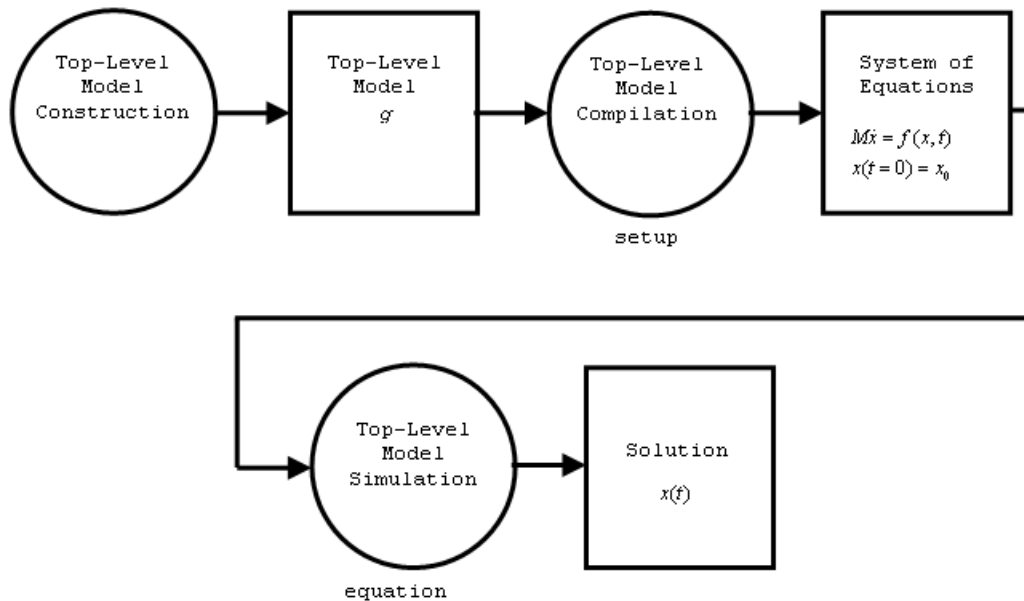
Component Instance Lifecycle

Phase	Steps
Top-Level Model Construction	**1** Invokes file name from MATLAB to construct component instance **2** Adds component instance to top-level model **3** Sets parameters on component instance **4** Connects component instance to other members of the top-level model
Top-Level Model Compilation	**5** Calls the `setup` function once for each component instance in the top-level model
Top-Level Model Simulation	**6** (Conceptually) calls the `equations` function for each component instance in the top-level model repeatedly throughout the simulation

Creating a New Physical Domain

In this section...
"When to Define a New Physical Domain" on page 1-13
"How to Define a New Physical Domain" on page 1-14

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 "Foundation Domain Types and Directory Structure" on page 5-2.

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 2-67).

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 2-7.

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 2-67.

For an example of a domain file, see "Declare a Mechanical Rotational Domain" on page 2-13.

Creating Custom Components

In this section...

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.

- Composite — A model that is constructed out of other blocks, connected in a certain way. An example of a composite, or 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 and composite models when your design requirements are not satisfied by the libraries of standard blocks provided with Simscape and its add-on products.

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 1-13.

How to Create a New Component

To create a new custom component, 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 the following 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 2-16.

- Structure — Declares the component connections for composite models. For more information, see "Creating Composite Components" on page 2-47.

- Equation — Declares the component equations for behavioral models. These equations may be conditional, and are applied throughout the simulation. For more information, see "Defining Component Equations" on page 2-21.

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 2-67 and "Attribute Lists" on page 2-76.

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 "Generate Custom Block Libraries 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.

2

Creating Custom Components and Domains

Declaring Domains and Components

Declaration Section Purpose

domain *domain_name*
Declaration section

component *component_name*
Declaration section
Setup section
Structure section
Equation section

Both domain and component files contain a declaration section:

- The declaration section of a domain file is where you define the Through and Across variables for the domain. You can also define the domain-wide parameters, if needed.

- The declaration section of a component file is where you define all the variables, parameters, nodes, inputs, and outputs that you need to describe the connections and behavior of the component. These are called member declarations.

  In order to use a variable, parameter, and so on, in the setup and equation sections of a component file, you have to first define it in the declaration section.

Definitions

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

Term	Definition
Member	• 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 *member class*. Additionally, members may have some specific attributes.
Member class	• 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`, `components`. The `components` member class, not to be confused with the `component` model class, is discussed in "Declaring Member Components" on page 2-47. • Two members may have the same type, but be of different member classes. For example, a parameter and an input may 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

Member Class	Applicable Model Classes	MATLAB Class of Expression	Expression Meaning	Writable
nodes	component	Instance of a node associated with a domain	Type of domain	No
components	component	Instance of a component class	Member component included in a composite model (see "Declaring Member Components" on page 2-47)	No

---

Note When a member is writable, it means that it can be assigned to in the setup function. nodes and components 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 "Declare a Mechanical Rotational Domain" on page 2-13 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 N*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 "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 2-77).

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 2-67.

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 "Generate Custom Block Libraries 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 "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,
where `value` can be a scalar, vector, or matrix. For a vector or a matrix, all
signals have the same unit.

Specifying an optional comment lets you control the port label and location
in the block icon. For more information, see "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
```

The next example declares an output port v as a 3-by-3 matrix of linear velocities.

```
outputs
  v = {zeros(3), 'm/s'};
end
```

Declare 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 "Foundation Domain Types and Directory Structure" on page 5-2.

Declare 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.
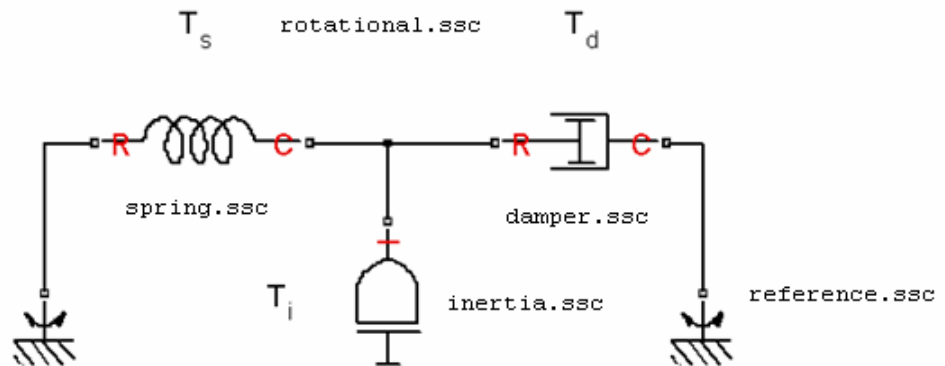
The domain is declared in a file named rotational.ssc (see "Declare a Mechanical Rotational Domain" on page 2-13). The following file, named spring.ssc, declares a component called spring. The component contains:

- Two rotational nodes, r and c (for rod and case, respectively)

- Parameter k, with a default value of 10 N*m/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 — Spring" on page 2-56.

Defining Component Setup

In this section...

Setup Section Purpose

component *component_name*

Declaration section

Setup section

Structure section
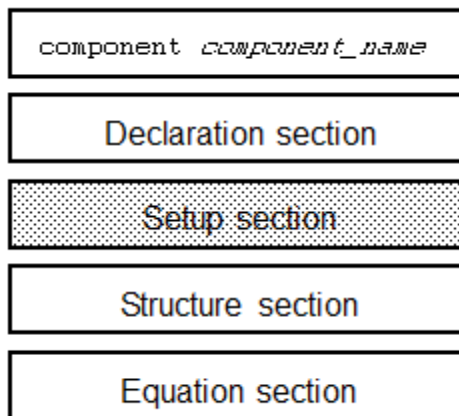
Equation section

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 "Persistent Variables" and "Global 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' );
         MyParam = MaxValue;
      end
   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
      Speed = { 10, 'm/s' };
   end
  [...]
   parameters
      InCollision = 0;  % Specifies whether bodies are in collision
   end
   [...]
   function setup
      if InCollision > 0
         Speed = { 0, 'm/s' };   % Speed(t = 0) = 0 because bodies are in collision
      end
   end
   [...]
end
```

Note Variables are also strongly typed. The initial value you assign to the variable must have the same data type and compatible unit as the variable. 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

Equation Section Purpose

The equation section of a Simscape file follows the declaration, setup, and structure 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.

A Simscape language equation consists of two expressions connected with the == operator. Unlike the regular assignment operator (=), the == operator specifies continuous mathematical equality between the two expressions (for more information, see "Specifying Mathematical Equality" on page 2-23). The equation expressions may be constructed from any of the identifiers defined in the model declaration. You can also access global simulation time from the equation section using the `time` function.

Equations in Simscape Language

- "Simple Algebraic System" on page 2-22
- "Using Simulation Time in Equations" on page 2-23

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
```

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 on page 4-29).

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.

You can nest `if` statements, for example:

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

Restrictions

- Every `if` requires an `else`.

- The total number of equation expressions, their dimensionality, and their order must be the same for every branch of the `if-elseif-else` statement. However, this rule does not apply to the `assert` expressions, because they are not included in the expression count for the branch.

- Every branch of the `if-elseif-else` statement must define the same variable in terms of others. For example, you can design a hydraulic orifice with `if-else` branches for turbulent and laminar flow, where each branch defines flow rate in terms of pressure. However, a conditional expression similar to the following

  ```
  if x > 0
      i == 0;
  else
      v == 0;
  end
  ```

  is forbidden.

Example

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

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

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 2-30
- "Syntax Rules" on page 2-32
- "Nested `let` Expressions" on page 2-35
- "Conditional `let` Expressions" on page 2-37
- "Identifier List in the Declarative Clause" on page 2-39

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:

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

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

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

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

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
```

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 that 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
```

Using Lookup Tables in Equations

You can use the tablelookup function in the equations section of the
Simscape file to interpolate input values based on a set of data points in a

one-dimensional or two-dimensional table. This functionality is similar to that of the Simulink and Simscape Lookup Table blocks. It allows you to incorporate table-driven modeling directly in your custom block, without the need of connecting an external Lookup Table block to your model.

The following example implements mapping temperature to pressure using a one-dimensional lookup table.

```
component TtoP
 inputs
   u = {0, 'K'}; % temperature
 end
 outputs
   y = {0, 'Pa'}; % pressure
 end
 parameters (Size=variable)
   xd = {[100 200 300 400] 'K'};
   yd = {[1e5 2e5 3e5 4e5] 'Pa'};
 end
 equations
   y == tablelookup(xd, yd, u, interpolation=linear, extrapolation=nearest);
 end
end
```

`xd` and `yd` are declared as variable-size parameters with units. This enables the block users to provide their own data sets when the component is converted to a custom block, and also to select commensurate units from the drop-downs in the custom block dialog box. The next illustration shows the dialog box of the custom block generated from this component.
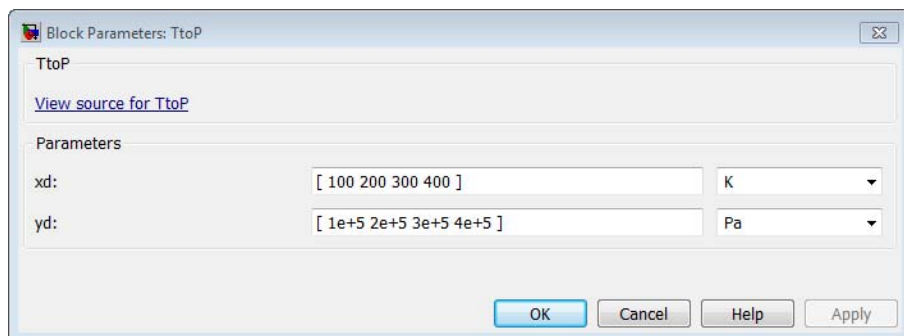
> **Note** Currently, you can not use variable-size parameters in the `equations` section outside of the `tablelookup` function.

> To avoid repeating the same variable-size parameter declarations in each component that needs to use them in its `tablelookup` function, you can declare variable-size domain parameters and propagate them to components for interpolation purposes. For more information, see "Propagation of Domain Parameters" on page 2-67.

The following rules apply to the one-dimensional arrays `xd` and `yd`:

- The two arrays must be of the same size.

- For cubic or spline interpolation, each array must contain at least three values. For linear interpolation, two values are sufficient.

- The `xd` values must be strictly monotonic, either increasing or decreasing.

If these rules are violated by...	You get an error...
The block author, in the component Simscape file	At build time, when running `ssc_build` to convert the component to a custom block
The block user, when entering values in the block dialog box	At simulation time, when attempting to simulate the model containing the custom block

The TtoP component uses linear interpolation for values within the table range, but outputs the nearest value of `yd` for out-of-range input values. The following illustration shows a block diagram, where the custom TtoP block is used with a linear input signal changing from 0 to 1000, and the resulting output.
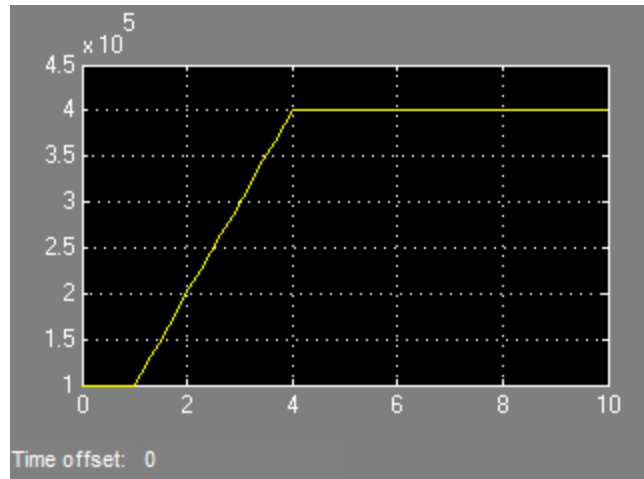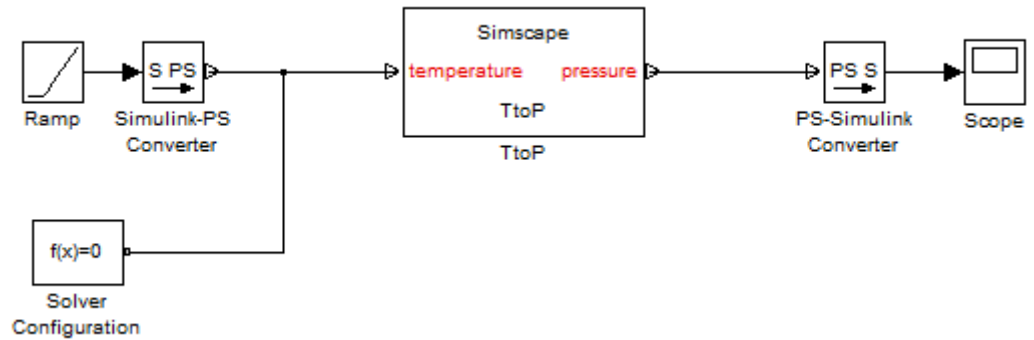
See the `tablelookup` reference page for syntax specifics and more examples.

Programming Run-Time Errors and Warnings

Use the `assert` construct to implement run-time error and warning messages for a custom block. In the component file, you specify the condition to be evaluated, as well as the error message to be output if this condition is violated. When the custom block based on this component file is used in a model, it will output this message if the condition is violated during simulation. The `Warn` attribute of the `assert` construct specifies whether

simulation stops when the predicate condition is violated, or continues with a warning.

The following component file implements a variable resistor, where input physical signal R supplies the resistance value. The `assert` construct checks that this input signal is greater than or equal to zero:

```
component MyVariableResistor
% Variable Resistor
% Models a linear variable resistor. The relationship between voltage V
% and current I is V=I*R where R is the numerical value presented at the
% physical signal port R. If this signal becomes negative, simulation
% errors out.
%

  inputs
    R = { 0.0, 'Ohm' };
  end

  nodes
    p = foundation.electrical.electrical; % +:left
    n = foundation.electrical.electrical; % -:right
  end

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

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

  equations
      assert( R >= 0, 'Negative resistance is not modeled' );
      v == i*R;
  end

end
```

If a model contains this Variable Resistor block, and signal R becomes negative during simulation, then simulation stops and the Simulation Diagnostics window opens with a message similar to the following:

```
At time 3.200000, an assertion is triggered. Negative resistance is not modeled.
The assertion comes from:
Block path: dc_motor1/Variable Resistor
Assert location: between line: 29, column: 14 and line: 29, column: 18 in file:
C:/Work/libraries/+MySimscapeLibrary/+ElectricalElements/MyVariableResistor.ssc
```

The error message contains the following information:

- Simulation time when the assertion got triggered
- The *message* string (in this example, `Negative resistance is not modeled`)
- An active link to the block that triggered the assertion. Click the `Block path` link to highlight the block in the model diagram.
- An active link to the assert location in the component source file. Click the `Assert location` link to open the Simscape source file of the component, with the cursor at the start of violated predicate condition. For Simscape protected files, the `Assert location` information is omitted from the error message.

See the `assert` reference page for syntax specifics and more examples.

Working with Physical Units in Equations

In Simscape language, you declare members (such as parameters, variables, inputs, and outputs) as value with unit, and the equations automatically handle all unit conversions.

However, empirical formulae often employ noninteger exponents where the base is either unitless or in known units. When working with these types of formulae, convert the base to a unitless value using the `value` function and then reapply units if needed.

For example, the following formula gives the pressure drop, in Pa, in terms of flow rate, in m^3/s:

```
p == k * q^1.023
```

where *p* is pressure, *q* is flow rate and k is some unitless constant. To write this formula in Simscape language, use:

```
p == { k * value(q, 'm^3/s')^1.023, 'Pa' }
```

This approach works regardless of the actual units of *p* or *q*, as long as they are commensurate with pressure and volumetric flow rate, respectively. For example, the actual flow rate can be in gallons per minute, the equation will still work and handle the unit conversion automatically.

Creating Composite Components

About Composite Components

A composite component is constructed out of other components. To create a composite component, you have to list the names of the member (constituent) components and then specify how the ports of the member components are connected to each other and to the external ports of the composite component. You also specify which parameters of the member components are to be visible, and therefore adjustable, in the block dialog box of the composite component.

In certain ways, this functionality is similar to creating a subsystem in a Simulink block diagram, however there are important differences. Simscape language is a textual environment, and therefore you cannot "look under mask" and see a graphical representation of the underlying component connections. At the same time, the textual environment is a very powerful tool for modeling complex modular systems that consist of multiple interconnected member components.

Declaring Member Components

A `components` declaration block begins with a `components` keyword and is terminated by an `end` keyword. This block contains declarations for member components included in the composite component. A `components` declaration block must have its `Hidden` attribute value set to `true` (for more information on member attributes, see "Attribute Lists" on page 2-76).

When declaring a member component, you have to associate it with an existing component file, either in the Simscape Foundation libraries or in your custom package. You need to refer to the component name using the full path starting with the top package directory. For more information on

packaging your Simscape files, see "Generate Custom Block Libraries from Simscape Component Files" on page 3-2.

The following example includes a Rotational Spring block from the Simscape Foundation library in your custom component:

```
components(Hidden=true)
    rot_spring = foundation.mechanical.rotational.spring;
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 component file spring.ssc.

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

```
components(Hidden=true)
    rot_spring = MechanicalElements.spring;
end
```

Once you declare a member component, use its identifier (in the preceding examples, rot_spring) to refer to its parameters, variables, nodes, inputs, and outputs. For example, rot_spring.spr_rate refers to the **Spring rate** parameter of the Rotational Spring block.

Parameterizing Composite Components

Composite component parameters let you adjust the desired parameters of the underlying member components from the top-level block dialog box when building and simulating a model.

Specify the composite component parameters by declaring a corresponding parameter in the top-level parameters declaration block, and then assigning it to the desired parameter of a member component. The declaration syntax is the same as described in "Declaring Component Parameters" on page 2-8.

For example, the following code includes a Foundation library Resistor block in your custom component file, with the ability to control the resistance at the top level and a default resistance of 10 Ohm:

```
component MyCompositeModel
[...]
  parameters
     p1 = {10, 'Ohm'};
     [...]
  end
  components(Hidden=true)
     r1 = foundation.electrical.elements.resistor(R = p1);
     [...]
  end
[...]
end
```

You can establish the connection of the top-level parameter with a member component parameter either in the components declaration block, or later, in the setup section. The following code is equivalent to the example above:

```
component MyCompositeModel
[...]
  parameters
     p1 = {10, 'Ohm'};
     [...]
  end
  components(Hidden=true)
     r1 = foundation.electrical.elements.resistor;
     ...
  end
   [...]
   function setup
      r1.R = p1;
   end
   [...]
end
```

> **Note** In case of conflict, assignments in the setup section override those made in the declaration section.

Components are instantiated using default parameter values in the declaration section before setup is run. Therefore, if you make adjustments to the parameters in the setup section, use a subsequent setup section assignment to establish proper connection between the top-level parameter with a member component parameter, as shown in the following example:

```
component RC
  nodes
    p = foundation.electrical.electrical; % :right
    n = foundation.electrical.electrical; % :left
  end
  parameters
    R = {1 , 'Ohm'}; % Resistance
    tc = {1 , 's'};  % RC time constant
  end
  parameters(Hidden=true)
    C = {1 , 'F'};
  end
  components(Hidden=true)
    c1 = foundation.electrical.elements.capacitor(c=C);
    r1 = foundation.electrical.elements.resistor(R=R);
  end
  function setup
    C = tc/R;
    c1.c = C; % This assignment ensures correct operation
  end
  connections
    connect(c1.p, p);
    connect(c1.n, r1.p);
    connect(r1.n, n);
  end
end
```

You do not have to assign all the parameters of member blocks to top-level parameters. If a member block parameter does not have a corresponding

top-level parameter, the composite model uses the default value of this parameter, specified in the member component. You can also use the `setup` section of the composite component to override the default value of the member component parameter and set it to a value applicable to your composite model.

Specifying Component Connections

The structure section of a Simscape file follows the declaration and setup sections. It is executed once during compilation. This section contains information on how the constituent components' ports are connected to one another, as well as to the external inputs, outputs, and nodes of the top-level component.
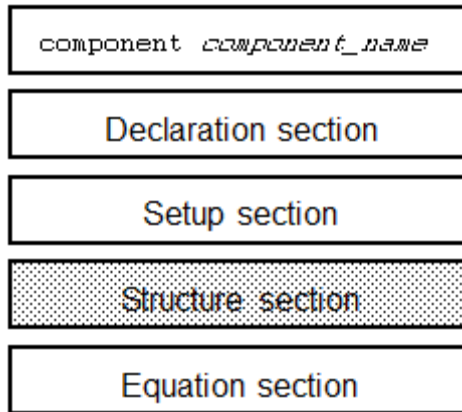
The structure section begins with a `connections` keyword and is terminated by an `end` keyword. This `connections` block contains a set of `connect` constructs, which describe both the conserving connections (between `nodes`) and the physical signal connections (between the `inputs` and `outputs`).

In the following example, the custom component file includes the Foundation library Voltage Sensor and Electrical Reference blocks and specifies the following connections:

- Positive port of the voltage sensor to the external electrical conserving port + of the composite component

- Negative port of the voltage sensor to ground

- Physical signal output port of the voltage sensor to the external output of the composite component, located on the right side of the resulting block icon
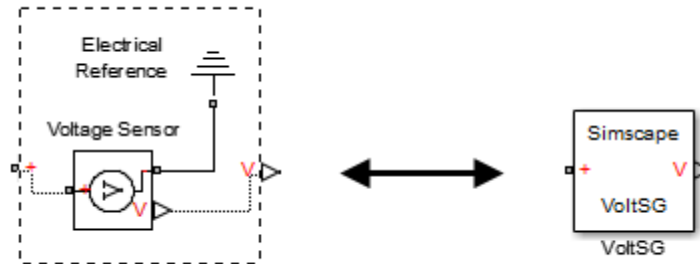

```
component VoltSG
  nodes
    p = foundation.electrical.electrical; % +
  end
  outputs
    Out = { 0.0, 'V' }; % V:right
  end
  components(Hidden=true)
    VoltSensor = foundation.electrical.sensors.voltage;
    Grnd = foundation.electrical.elements.reference;
  end
  connections
    connect(p, VoltSensor.p);
    connect(Grnd.V, VoltSensor.n);
    connect(VoltSensor.V, Out);
  end
end
```

In this example, the first two `connect` constructs specify conserving connections between electrical nodes. The third `connect` construct is a physical signal connection. Although these constructs look similar, their syntax rules are different.

Conserving Connections

For conserving connections, the `connect` construct can have two or more arguments. For example, the connections in the following example

```
connections
    connect(R1.p, R2.n);
    connect(R1.p, R3.p);
end
```

can be replaced with

```
connections
    connect(R1.p, R2.n, R3.p);
end
```

The order of arguments does not matter. The only requirement is that the nodes being connected are all of the same type (that is, are all associated with the same domain).

In the following example, the composite component consists of three identical resistors connected in parallel:

```
component ParResistors
  nodes
     p = foundation.electrical.electrical;
     n = foundation.electrical.electrical;
  end
  parameters
    p1 = {3 , 'Ohm'};
  end
  components(Hidden=true)
    r1 = foundation.electrical.elements.resistor(R=p1);
    r2 = foundation.electrical.elements.resistor(R=p1);
    r3 = foundation.electrical.elements.resistor(R=p1);
  end
  connections
    connect(r1.p, r2.p, r3.p, p);
    connect(r1.n, r2.n, r3.n, n);
  end
end
```

Physical Signal Connections

Physical signal connections are directional, therefore the connect construct has the following format:

```
connect(s, d);
```

where s is the signal source port and d is the destination port.

There can be more than one destination port connected to the same source port:

```
connect(s, d1, d2);
```

The source and destination ports belong to the inputs or outputs member classes. The following table lists valid source and destination combinations.

Source	Destination
External input port of composite component	Input port of member component
Output port of member component	Input port of member component
Output port of member component	External output port of composite component

For example, consider the following block diagram.
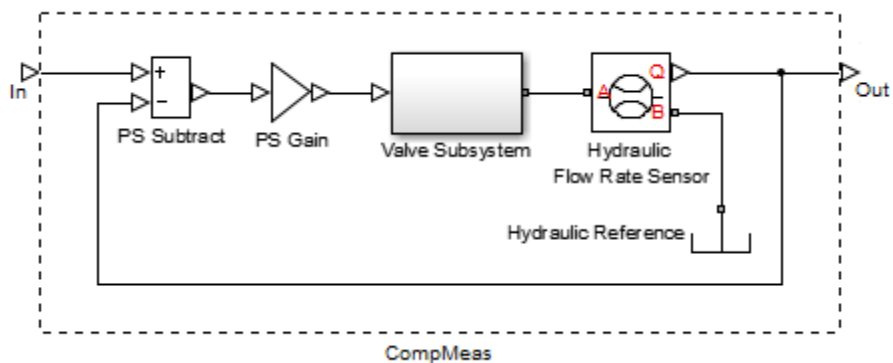
It represents a composite component `CompMeas`, which, in turn, contains a composite component `Valve Subsystem`, as well as several Foundation library blocks. The Simscape file of the composite component would specify the equivalent signal connections with the following constructs.

Construct	Explanation
`connect(In, subt.I1);`	Connects port `In` to the input port + of the PS Subtract block. Illustrates connecting an input port of the composite component to an input port of a member component.
`connect(subt.O, gain.I);`	Connects the output port of the PS Subtract block to the input port of the PS Gain block. Illustrates connecting an output port of a member component to an input port of another member component at the same level.
`connect(fl_rate.Q, subt.I2, Out);`	Connects the output port `Q` of the Hydraulic Flow Rate Sensor block to the input port - of the PS Subtract block and to the output port `Out` of the composite component. Illustrates connecting a single source to multiple destinations, and also connecting an output port of a member component to an output port of the enclosing composite component.

Also notice that the output port of the PS Gain block is connected to the input port of the Valve Subsystem composite block (another member component at the same level). Valve Subsystem is a standalone composite component, and therefore if you connect the output port of the PS Gain block to an input port of one of the member components inside the Valve Subsystem, that would violate the causality of the physical signal connections (a destination port cannot be connected to multiple sources).

Putting It Together — Complete Component Examples

Mechanical Component — 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 N*m/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 * theta`, that is, torque equals spring deformation times spring rate

- `w = 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 — 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*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*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 — 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 — 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
```

Composite Component — DC Motor

In the Permanent Magnet DC Motor example, the DC Motor block is implemented as a masked subsystem.
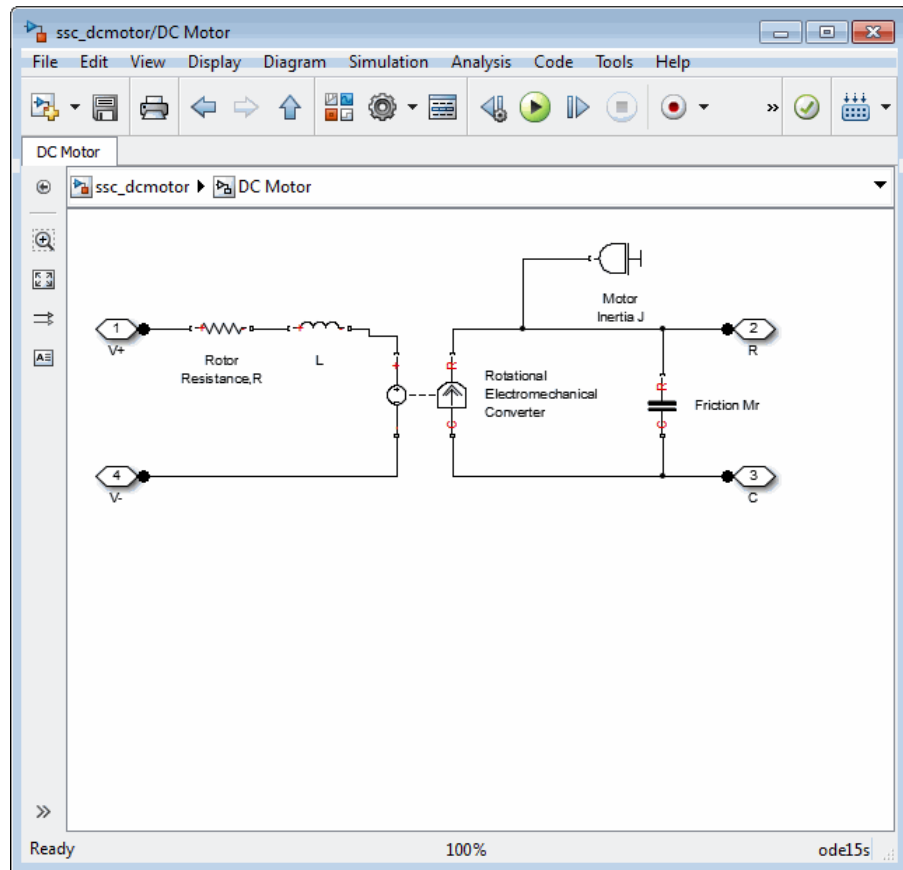
The following code implements the same model by means of a composite component, called DC Motor. The composite component uses the components from the Simscape Foundation library as building blocks, and connects them as shown in the preceding block diagram.

```
component DC_Motor
% DC Motor
% This block models a DC motor with an equivalent circuit comprising a
% series connection of a resistor, inductor and electromechanical converter.
% Default values are as for the DC Motor Simscape example, ssc_dcmotor.

nodes
```

```
    p = foundation.electrical.electrical;              % +:left
    n = foundation.electrical.electrical;              % -:left
    R = foundation.mechanical.rotational.rotational;   % R:right
    C = foundation.mechanical.rotational.rotational;   % C:right
end

parameters
    rotor_resistance    = { 3.9, 'Ohm' };            % Rotor Resistance
    rotor_inductance    = { 12e-6, 'H' };            % Rotor Inductance
    initial_current     = { 0, 'A' };                % Initial current for Rotor Inductor
    motor_inertia       = { 0.01, 'g*cm^2' };        % Inertia
    init_velocity       = { 0, 'rad/s' };            % Initial velocity
    breakaway_torque    = { 0.02e-3, 'N*m' };        % Breakaway friction torque
    coulomb_torque      = { 0.02e-3, 'N*m' };        % Coulomb friction torque
    viscous_coeff       = { 0, 'N*m*s/rad' };        % Viscous friction coefficient
    velocity_threshold  = { 0.1, 'rad/s' };          % Linear region velocity threshold
    back_emf_constant   = { 0.072e-3, 'V/rpm' };     % Back EMF constant
end

components(Hidden=true)
    rotorResistor                 = foundation.electrical.elements.resistor(R = rotor_resistance);
    rotorInductor                 = foundation.electrical.elements.inductor(l = rotor_inductance,
                                       i0 = initial_current);
    rotationalElectroMechConverter = foundation.electrical.elements.rotational_converter(K =
                                       back_emf_constant);
    friction                      = foundation.mechanical.rotational.friction(brkwy_trq =
                                       breakaway_torque, Col_trq = coulomb_torque,
                                       visc_coef = viscous_coeff, vel_thr = velocity_threshold);
    motorInertia                  = foundation.mechanical.rotational.inertia(inertia = motor_inertia,
                                       initial_velocity = init_velocity);
end

connections
    connect(p, rotorResistor.p);
    connect(rotorResistor.n, rotorInductor.p);
    connect(rotorInductor.n, rotationalElectroMechConverter.p);
    connect(rotationalElectroMechConverter.n, n);
    connect(rotationalElectroMechConverter.R, friction.R, motorInertia.I, R);
    connect(rotationalElectroMechConverter.C, friction.C, C);
end
```

2-63

```
end
```

The declaration section of the composite component starts with the `nodes` section, which defines the top-level connection ports of the resulting composite block:

- Two electrical conserving ports, + and -, on the left side of the block

- Two mechanical rotational conserving ports, `R` and `C`, on the right side of the block
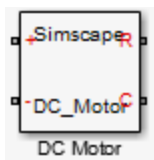
The `parameters` declaration block lists all the parameters that will be available in the composite block dialog box.
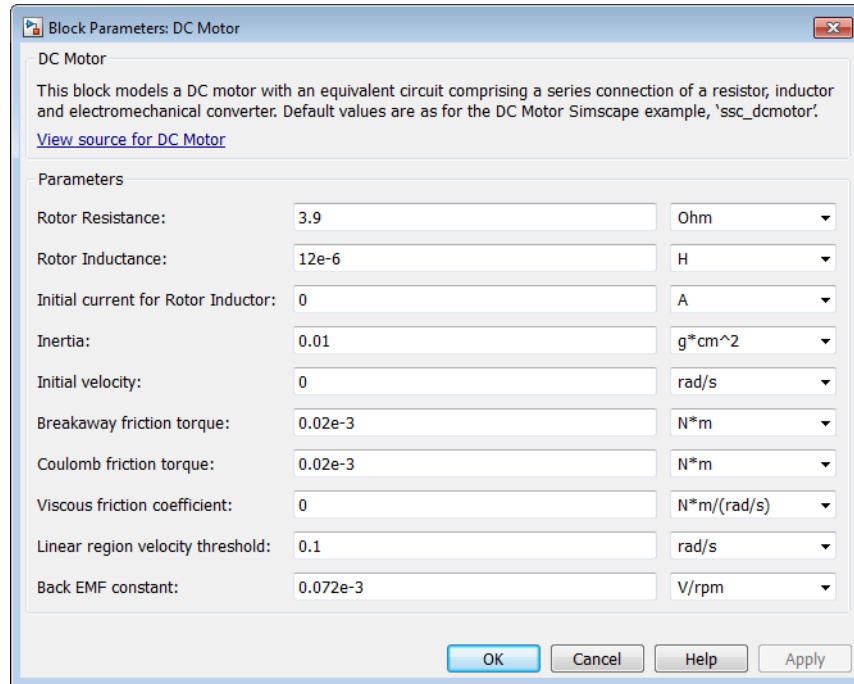
The `components` block declares all the member (constituent) components, specifying their complete names starting from the top-level package directory. This example uses the components from the Simscape Foundation library:

- Resistor

- Inductor

- Rotational Electromechanical Converter

- Rotational Friction

- Inertia

The `components` block also links the top-level parameters, declared in the `parameters` declaration block, to the parameters of underlying member components. For example, the **Rotor Resistance** parameter of the composite block (`rotor_resistance`) corresponds to the **Resistance** parameter (`R`) of the Resistor block in the Foundation library.

You do not have to link all the parameters of member blocks to top-level parameters. For example, the Rotational Friction block in the Foundation library has the **Transition approximation coefficient** parameter, which is not mapped to any parameter at the top level. Therefore, the composite model always uses the default value of this parameter specified in the Rotational Friction component, 10 rad/s.

The `connections` block defines the connections between the nodes (ports) of the member components, and their connections to the top-level ports of the resulting composite block, declared in the `nodes` declaration block of the composite component:

- Positive electrical port `p` of the composite component is connected to the positive electrical port `p` of the Resistor

- Negative electrical port `n` of the Resistor is connected to the positive electrical port `p` of the Inductor

- Negative electrical port `n` of the Inductor is connected to the positive electrical port `p` of the Rotational Electromechanical Converter

- Negative electrical port `n` of the Rotational Electromechanical Converter is connected to the negative electrical port `n` of the composite component

- Mechanical rotational port `R` of the composite component is connected to the following mechanical rotational ports: `R` of the Rotational Electromechanical Converter, `R` of the Rotational Friction, and `I` of the Inertia

- Mechanical rotational port `C` of the composite component is connected to the following mechanical rotational ports: `C` of the Rotational Electromechanical Converter and `C` of the Rotational Friction

These connections are the textual equivalent of the graphical connections in the preceding block diagram.

Working with Domain Parameters

In this section...

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 2-76.

For a complete example of building a custom block library based on this domain definition and using propagation of domain parameters in a simple circuit, see "Custom Library with Propagation of Domain Parameters" on page 2-69.

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 2-67. 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 2-67. 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.

Custom Library with Propagation of 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 2-67.

```
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 2-68. 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 2-69. 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 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
```
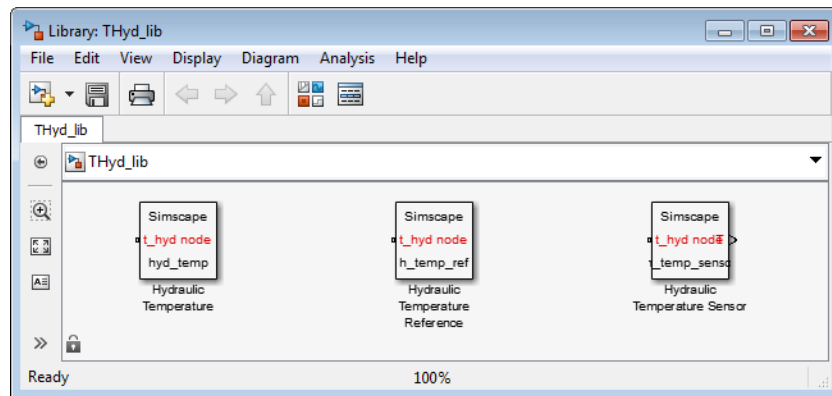

8 Create a new Simscape model. To do this, type:

```
ssc_new
```

This command creates a new model, prepopulated with the following blocks:
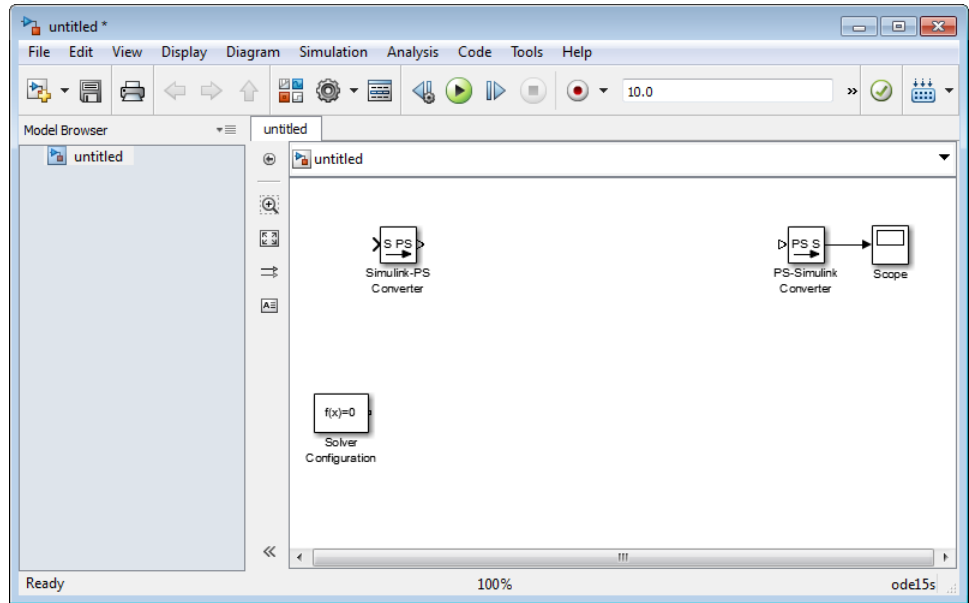
9 Delete the Simulink-PS Converter block, because our model is not going to have any Simulink input signals.

10 Drag the Hydraulic Temperature, Hydraulic Temperature Sensor, and Hydraulic Temperature Reference blocks from THyd_lib and connect them as follows:
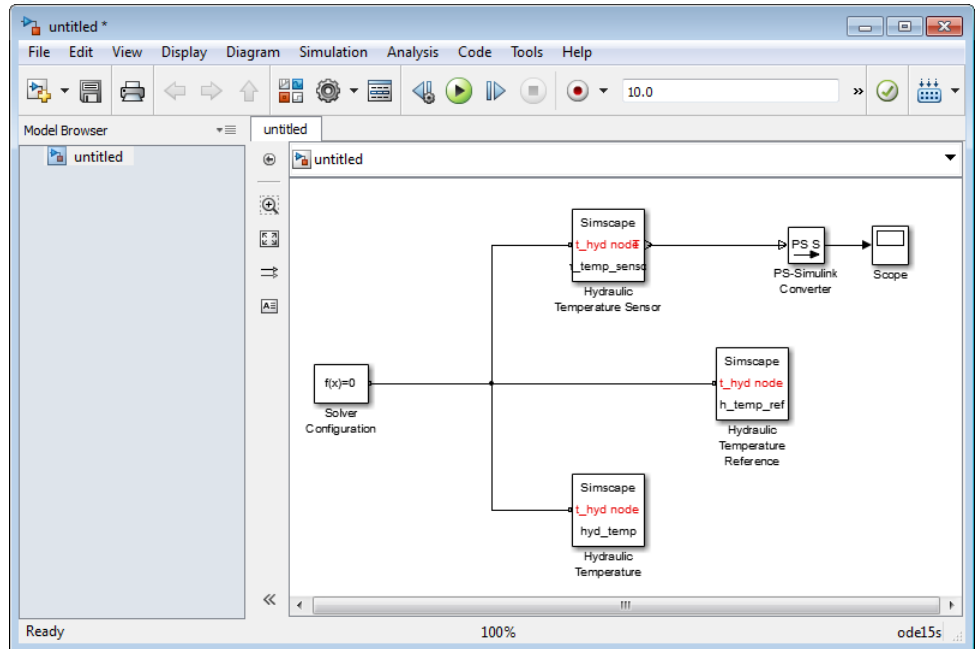
11 Simulate the model and notice that the scope displays the value of the domain temperature parameter, as it is defined in the hyd_temp.ssc file, 333 K.

12 Double-click the Hydraulic Temperature block. Change the value of the **Fluid temperature** parameter to 363 K.
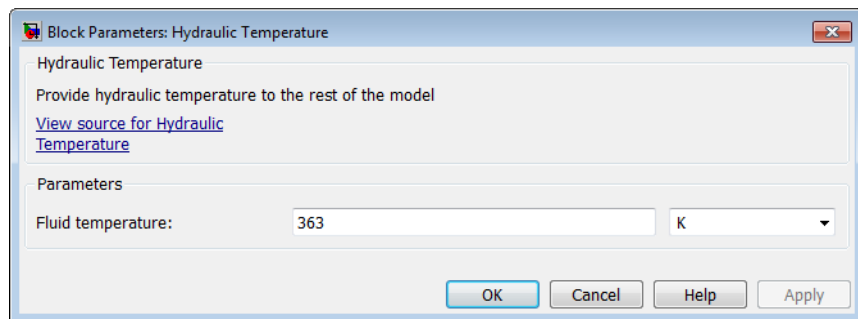
13 Simulate the model again and notice that the scope now displays the new value of the domain temperature parameter.

Attribute Lists

In this section...
"Attribute Types" on page 2-76
"Model Attributes" on page 2-76
"Member Attributes" on page 2-77

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 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 2-67.
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 ( Propagation = source ) MyParameterSource
  % 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.
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 2-7.
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 2-9.

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.

Note If you are using subclassing with composite components, there is a limitation. You cannot override a parameter value for a member component of a base class by using 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 create the base class `ElectricalBranch.ssc`, which defines an electrical branch with positive and negative external nodes, initial current and voltage:

```
component ElectricalBranch
  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
% Ideal Capacitor
  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
```

The subclass component inherits the p and n nodes, as well as the i
and v variables with initial values, from the base class. This way, the
Capacitor.ssc file contains only parameters, setup, and equations specific
to the capacitor.

3

Simscape File Deployment

Generate Custom Block Libraries from Simscape Component Files

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 "Packages Create Namespaces" 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` 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 "Build 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 Simulink model file called `SimscapeCustomBlocks_lib` 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.

When to Rebuild the Custom Library

You need to rebuild the custom Simscape libraries:

- Whenever you modify the source files.

- For use on each platform. Textual component files are platform-independent, but Simscape blocks are not. If you (or your customers) run MATLAB on multiple platforms, generate a separate version of custom block libraries for each platform by running the `ssc_build` or `ssc_mirror` command on this platform.

• For use with each new version of Simscape software. Every time you or your customers upgrade to a new release, you or they have to rebuild the custom block libraries. For information on how to protect your proprietary source code when sharing the Simscape files with customers, see "Using Source Protection for Simscape Files" on page 3-3.

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. 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	libInfo.Name = *name*	*name* will be used as the name of the sublibrary (name of the Simulink subsystem corresponding to the sublibrary)	Package name	Ignored
Annotation	libInfo.Annotation = *annotation*	*annotation* will be displayed as annotation when you open the sublibrary. It can be any text	No annotation	Used in annotation for

Option	Usage	Description	Default	For Top-Level Package
		that you want to display in the sublibrary.	in the library	top-level library
ShowIcon	libInfo.ShowIcon = false	If there is no library icon file lib.*img*, 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 false.	true	Ignored
ShowName	libInfo.ShowName = true	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 showName to true to display the library name below the icon.	false	Ignored
Hidden	libInfo.Hidden = true	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 3-6.

Create a Custom Block Library

This example illustrates how you can convert a package of Simscape component files into a custom block library, containing sublibraries with customized names and appearance. It summarizes the techniques described in "Organizing Your Simscape Files" on page 3-3, "Converting Your Simscape Files" on page 3-4, and "Customizing the Library Name and Appearance" on page 3-6.

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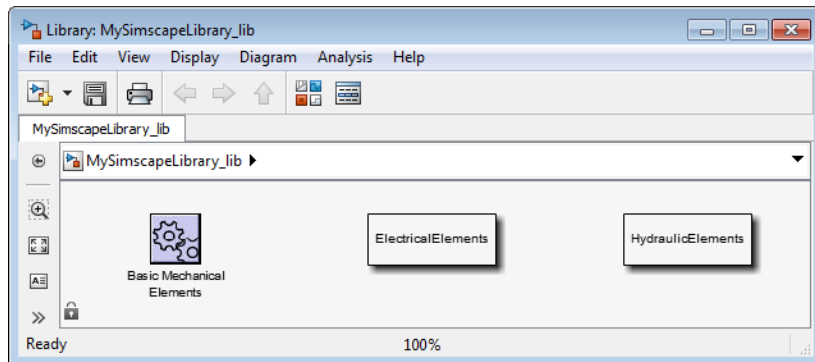
Notice that the sublibrary generated from the +MechanicalElements package is presented in its parent library with a customized icon and name (Basic Mechanical Elements).

If you double-click the Basic Mechanical Elements sublibrary, it opens as follows:
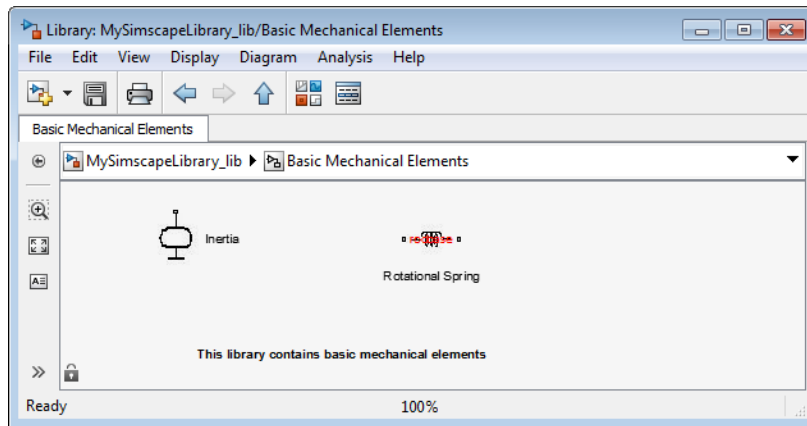
Customizing the Block Name and Appearance

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
```
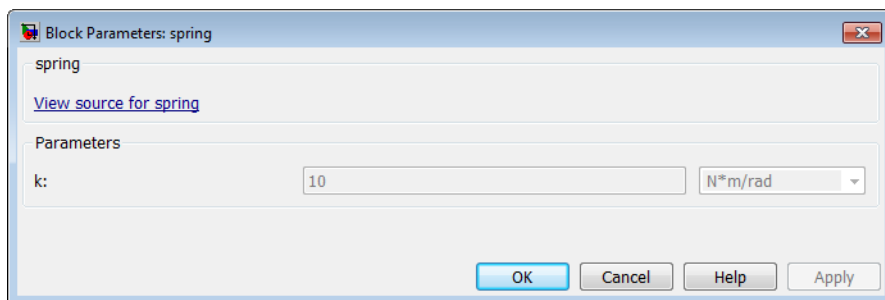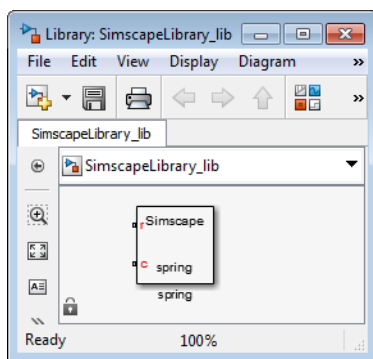

If you click the `View source for spring` link, the `spring.ssc` file opens in the MATLAB Editor window.

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.

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:
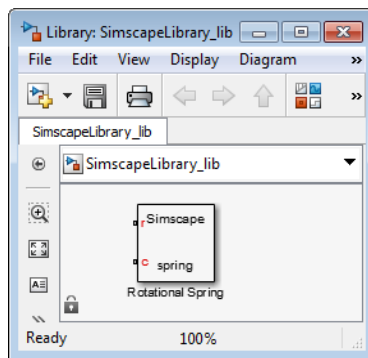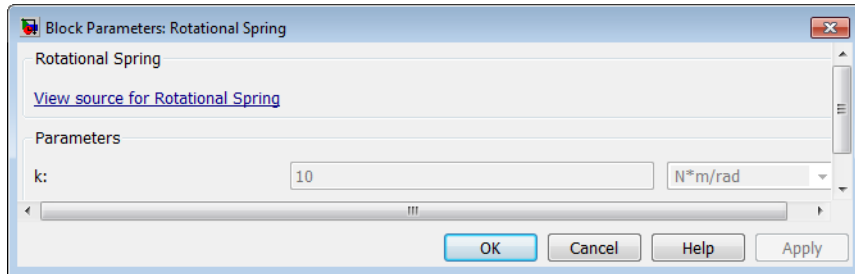
Describe the Block Purpose

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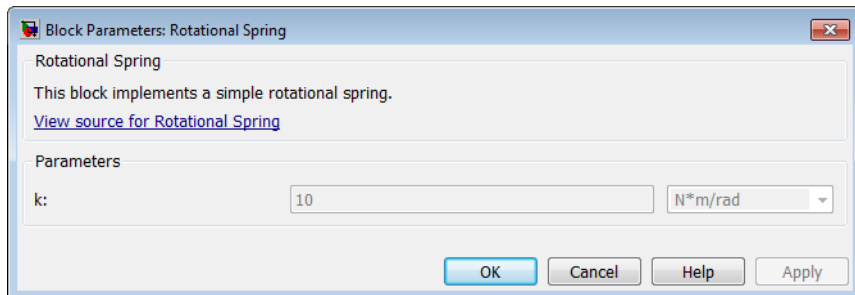
To create a paragraph break in the block description, use a blank commented line:

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

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:
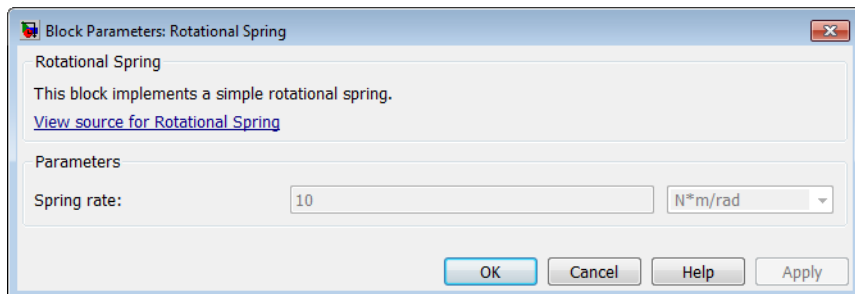
Customize the Names and Locations of the Block Ports

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.
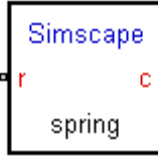
To control the port label and location in the block icon, add a comment immediately following the corresponding node, input, or output declaration. It can be on the same line or on a separate line. The comment must begin with the `%` character and be of the format `label:location`, where `label` is a 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
```nodes r = foundation.mechanical.rotational.rotational; c = foundation.mechanical.rotational.rotational; end```	
```nodes r = foundation.mechanical.rotational.rotational; % rod c = foundation.mechanical.rotational.rotational; % case end```	
```nodes r = foundation.mechanical.rotational.rotational; c = foundation.mechanical.rotational.rotational; % c:right end```	
```nodes r = foundation.mechanical.rotational.rotational; % rod c = foundation.mechanical.rotational.rotational; % case:right end```	

Syntax	Block Icon
```nodes r = foundation.mechanical.rotational.rotational; % rod c = foundation.mechanical.rotational.rotational; % :right end ```	
```nodes r = foundation.mechanical.rotational.rotational; % c = foundation.mechanical.rotational.rotational; % case:right end ```	

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

name	Component name
CustomName	Customized block name, specified as described in "Customize the Block Name" on page 3-13. Leading and trailing white spaces are removed.
scale	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.
rotation	Specifies whether the block icon rotates with the block: • 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.

Custom 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`. This example is an illustration of all the techniques described in "Customizing the Block Name and Appearance" on page 3-11.

```
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
```
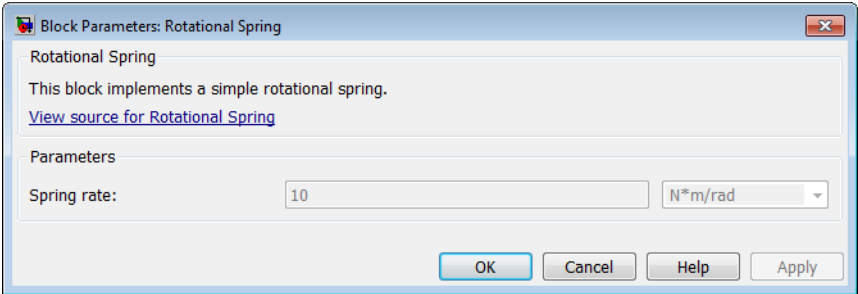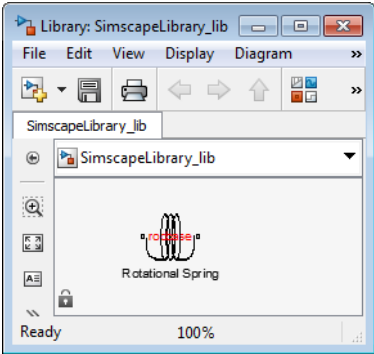
Checking File and Model Dependencies

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 "Analyze 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 **Analysis** > **Model Dependencies** > **Generate Manifest**. The Generate Model Manifest dialog box opens. For more information, see "Analyze Model Dependencies".

To check dependencies on Simscape blocks and files only, use the function simscape.dependency.model. For example, open the model dc_motor 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 — Basic Custom Block Library

In this section...

Getting Started

This case study explains how to build your own library of custom blocks based on component files. It uses an example 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 — Electrochemical Library" on page 3-33.

The example 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 example 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 "Generate Custom Block Libraries 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 library 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*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 library 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 example 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.
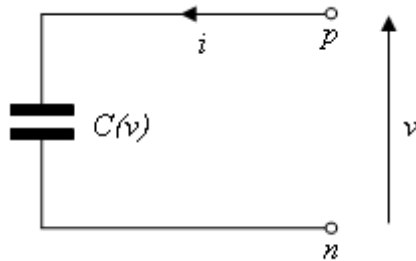
Ideal Ultracapacitor

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:

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

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

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

and hence

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

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 (`C0` and `Cv`) must be defined in the declaration section. For more information, see "Declaring Component Variables" on page 2-8and "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.
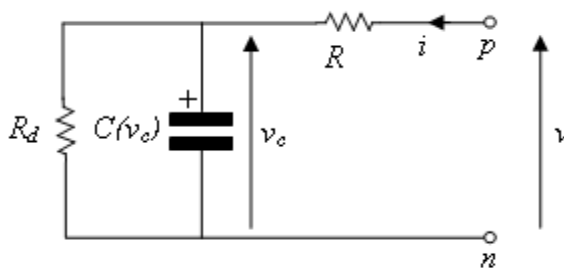
Ultracapacitor with Resistive Losses

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 = C0 + V*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
      CO = { 1, 'F' };    % Nominal capacitance CO at V=O
      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
      VO = { 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 CO <= O
         error( 'Nominal capacitance CO must be greater than zero' )
      end
      if R <= O
         error( 'Effective series resistance must be greater than zero' )
      end
      if Rd <= O
         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 = VO;
   end
   equations
      i == (CO + 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 == (CO + Cv*v)*v.der + vc/Rd;
```

3-31

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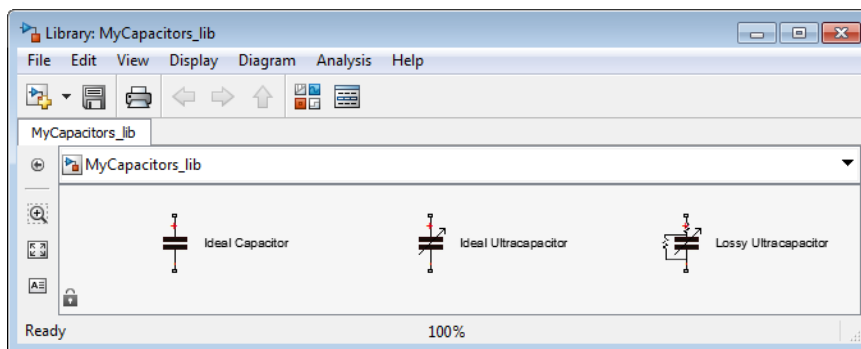
During the library build, if there is an image file in the directory with the same name as the Simscape component file, then this is used to define the icon for the block. For example, the Ideal Capacitor block defined by IdealCapacitor.ssc uses the IdealCapacitor.jpg to define its block icon. If you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

Case Study — Electrochemical Library

In this section...

Getting Started

This case study explores more advanced topics of building custom Simscape libraries. It uses an example 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 — Basic Custom Block Library" on page 3-26, before looking at this more advanced example.

The example 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 example 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 "Generate Custom Block Libraries 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].
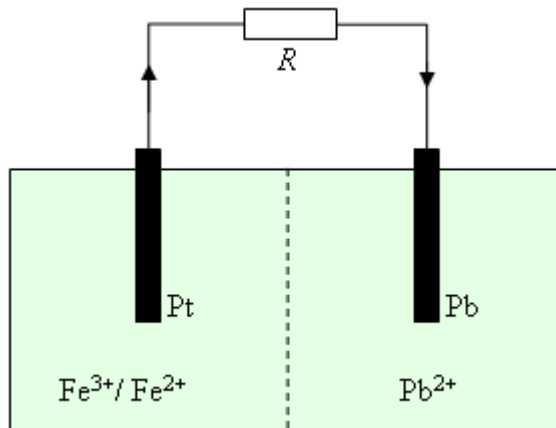
Electrochemical Battery Driving a Resistive Load R

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:

$$Pb \leftrightarrow Pb^{2+} + 2e^-$$

$$Fe^{2+} \leftrightarrow Fe^{3+} + e^-$$
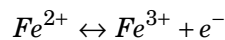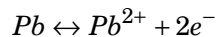
The current results in the lead being oxidized and the iron being reduced, with the overall reaction given by:

$$Pb + 2Fe^{3+} \leftrightarrow Pb^{2+} + 2Fe^{2+}$$
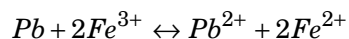
The chemical reaction can be modeled using the network concepts of Through and Across variables (for details, see "Basic Principles of Modeling Physical

3-35

Networks"). 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.
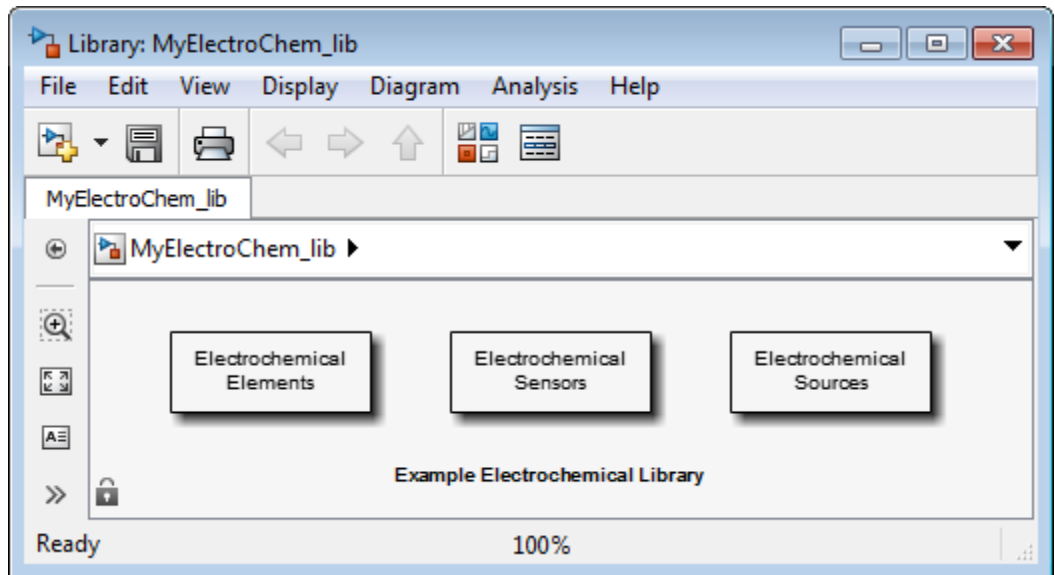
Defining a Reference Component

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 example 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 example library, this
source is not implemented.

Defining Measurement Components

Every domain requires both a Through and an Across
measurement block. In the example 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 mu 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 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:

$$\dot{n} = \nu$$

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

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

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 muO + log(n/(CO*M))*R*T where muO is the
% standard state oxidising potential, n is the number of moles of the ion,
% CO 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
    muO = {-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
        CO = {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/(CO*m_solvent))*R*T;
        else
            mu == mu0 + (log(n1/(CO*m_solvent)) + n/n1 - 1)*R*T;
        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 n1, 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 (CO) 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 n1 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:

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

where ν 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:

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

which can be rewritten as:

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

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`, 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 3-6.

Using the Custom Components to Build a Model

The Model Using a Customized Electrochemical Library example uses the electrochemical library to model a lead-iron battery. See the example 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.

4

Language Reference

tablelookup	Return value based on interpolating set of data points
through	Establish relationship between component variables and nodes
time	Access global simulation time
value	Convert variable or parameter to unitless value with specified unit conversion
variables	Define domain or component variables

Purpose   Establish relationship between component variables and nodes

Syntax   `across( variable1, node1.variableA, node2.variableB )`

Description   `across( variable1, node1.variableA, node2.variableB )` establishes the following relationship between the three arguments: `variable1` is assigned the value (`node1.variableA` `node2.variableB`). All arguments are variables. The first one is not associated with a node. The second and third must be associated with a node.

The following rules apply:

- All arguments must have consistent units.

- The second and third arguments do not need to be associated with the same domain. For example, one may be associated with a one-phase electrical domain, and the other with a 3-phase electrical.

- Either the second or the third argument may be replaced with `[]` to indicate the reference node.

Examples   If a component declaration section contains two electrical nodes, `p` and `n`, and a variable `v = { 0, 'V' };` specifying voltage, you can establish the following relationship in the setup section:

`across( v, p.v, n.v );`

This defines voltage `v` as an Across variable from node `p` to node `n`.

See Also   `through`

component

Purpose	Component model keywords

Syntax
```
component
nodes
inputs
outputs
parameters
variables
components
function setup
connections
equations
```

Description   `component` begins the component model class definition, which is terminated by an `end` keyword. Only blank lines and comments can precede `component`. You must place a component model class definition in a file of the same name with a file name extension of `.ssc`.

A component file consists of a declaration section, with one or more member declaration blocks, followed by setup and equation sections.

The declarations section may contain any of the following member declaration blocks.

`nodes` begins a nodes declaration block, which is terminated by an `end` keyword. This block contains declarations for all the component nodes, which correspond to the conserving ports of a Simscape block generated from the component file. Each node is defined by assignment to an existing domain. See "Declaring Component Nodes" on page 2-11 for more information.

`inputs` begins an inputs declaration block, which is terminated by an `end` keyword. This block contains declarations for all the inputs, which correspond to the input Physical Signal ports of a Simscape block generated from the component file. Each input is defined as a value with unit. See "Declaring Component Inputs and Outputs" on page 2-12 for more information.

`outputs` begins an outputs declaration block, which is terminated by an `end` keyword. This block contains declarations for all the outputs, which correspond to the output Physical Signal ports of a Simscape block generated from the component file. Each output is defined as a value with unit. See "Declaring Component Inputs and Outputs" on page 2-12 for more information.

`parameters` begins a component parameters declaration block, which is terminated by an `end` keyword. This block contains declarations for component parameters. Parameters will appear in the block dialog box when the component file is brought into a block model. Each parameter is defined as a value with unit. See "Declaring Component Parameters" on page 2-8 for more information.

`variables` begins a variables declaration block, which is terminated by an `end` keyword. This block contains declarations for all the variables associated with the component. Variables are internal to the component; they will not appear in a block dialog box when the component file is brought into a block model.

Variables can be defined either by assignment to an existing domain variable or as a value with unit. See "Declaring Component Variables" on page 2-8 for more information.

`components` begins a member components declaration block, which is terminated by an `end` keyword. This block, used in composite models only, contains declarations for member components included in the composite component. Each member component is defined by assignment to an existing component file. See "Declaring Member Components" on page 2-47 for more information.

`function setup` begins the setup section, which is terminated by an `end` keyword. This section relates inputs, outputs, and variables to one another by using `across` and `through` functions. It can also be used for validating parameters, computing derived parameters, and setting initial conditions. See "Defining Component Setup" on page 2-16 for more information.

`connections` begins the structure section, which is terminated by an `end` keyword. This section, used in composite models only, contains

information on how the constituent components' ports are connected to one another, and to the external inputs, outputs, and nodes of the top-level component. See "Specifying Component Connections" on page 2-51 for more information.

equations begins the equation section, which is terminated by an end keyword. This section contains the equations that define how the component works. See "Defining Component Equations" on page 2-21 for more information.

Table of Attributes

For component model attributes, as well as declaration member attributes, see "Attribute Lists" on page 2-76.

Examples

This file, named spring.ssc, defines a rotational spring.

```
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
```

See Also   domain

components

Purpose   Declare member components included in composite component

Syntax   components(Hidden=true) a = package_name.component_name; end

Description   components begins a components declaration block, which is terminated by an end keyword. This block, used in composite models only, contains declarations for member components included in the composite component. A components declaration block must have its Hidden attribute value set to true (for more information on member attributes, see "Attribute Lists" on page 2-76).

Each member component is defined by assignment to an existing component file. See "Declaring Member Components" on page 2-47 for more information.

The following syntax defines a member component, a, by associating it with a component file, component_name. package_name is the full path to the component file, starting with the top package directory. For more information on packaging your Simscape files, see "Generate Custom Block Libraries from Simscape Component Files" on page 3-2.

```
components(Hidden=true)
    a = package_name.component_name;
end
```

After you declare all member components, specify how their ports are connected to one another, and to the external inputs, outputs, and nodes of the top-level component. See "Specifying Component Connections" on page 2-51 for more information.

Once you declare a member component, you can use its parameters and variables in the setup section of the composite component file. If you want a parameter of the member component to be adjustable, associate it with the top-level parameter of the composite component. See "Parameterizing Composite Components" on page 2-48 for more information.

Examples   The following example includes a Rotational Spring block from the
Simscape Foundation library in your custom component:

```
components(Hidden=true)
    rot_spring = foundation.mechanical.rotational.spring;
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 component file spring.ssc.

Once you declare a member component, use its identifier (rot_spring)
to refer to its parameters, variables, nodes, inputs, and outputs,
as they are defined in the member component file. For example,
rot_spring.spr_rate refers to the **Spring rate** parameter of the
Rotational Spring block.

See Also   connections

parameters

connect

Purpose   Connect two or more component ports of the same type

Syntax
```
connect(n1, n2);
connect(s, d1);
```

Description   The connect constructs describe both the conserving connections (between nodes) and the physical signal connections (between the inputs and outputs). You can place a connect construct only inside the connections block in a composite component file.

For a conserving connection, the syntax is

```
connect(n1, n2);
```

The construct can have more than two arguments. n1, n2, n3, and so on are nodes declared in the composite component or in any of the member component files. The only requirement is that these nodes are all associated with the same domain. The order of arguments does not matter. The connect construct creates a physical conserving connection between all the nodes listed as arguments.

For a physical signal connection, the syntax is

```
connect(s, d1);
```

The construct can have more than two arguments. All arguments are inputs and outputs declared in the composite component or in any of the member component files. The first argument, s, is the source port, and the remaining arguments, d1, d2, d3, and so on, are destination ports. The connect construct creates a directional physical signal connection from the source port to the destination port or ports. For example,

```
connect(s, d1, d2);
```

means that source s is connected to two destinations, d1 and d2. A destination cannot be connected to more than one source. If a signal connect statement has more than one destination, the order of destination arguments (d1, d2, and so on) does not matter.

The following table lists valid source and destination combinations.

Source	Destination
External input port of composite component	Input port of member component
Output port of member component	Input port of member component
Output port of member component	External output port of composite component

If a member component is itself a composite component, the connect constructs can only access its external nodes, not the internal nodes of its underlying members. For example, consider the following diagram.
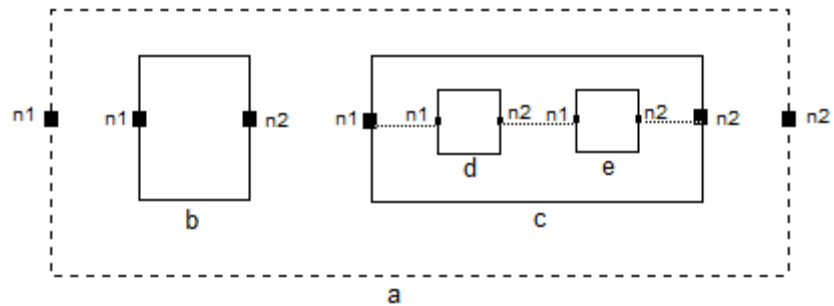
You are defining a composite component a, which consists of member components b and c. Component c is in turn a composite component containing members d and e. Each component has nodes n1 and n2.

The following constructs are legal:

```
connect(n1, c.n1);
```

```
connect(b.n1, c.n1);
```

However, the following constructs

```
connect(n1, c.d.n1);
```

connect

```
connect(b.n1, c.d.n1);
```

are illegal because they are trying to access an underlying member
component within the member component c.

Examples   In the following example, the composite component consists of three
identical resistors connected in parallel:

```
component ParResistors
  nodes
     p = foundation.electrical.electrical;
     n = foundation.electrical.electrical;
  end
  parameters
    p1 = {3 , 'Ohm'};
  end
  components(Hidden=true)
    r1 = foundation.electrical.elements.resistor(R=p1);
    r2 = foundation.electrical.elements.resistor(R=p1);
    r3 = foundation.electrical.elements.resistor(R=p1);
  end
  connections
    connect(r1.p, r2.p, r3.p, p);
    connect(r1.n, r2.n, r3.n, n);
  end
end
```

See Also   connections

"Specifying Component Connections" on page 2-51

Purpose   Define connections for member component ports in composite component

Syntax   connections connect(a, b); end

Description connections begins the structure section in a composite component
file; this section is terminated by an end keyword. It is executed
once during compilation. The structure section is located between
the setup and equation sections of the component file. This section
contains information on how the constituent components' ports are
connected to one another and to the external inputs, outputs, and
nodes of the top-level component. All member components declared in
the components declaration block are available by their names in the
structure section.

The connections block contains a set of connect constructs, which
describe both the conserving connections (between nodes) and the
physical signal connections (between the inputs and outputs). To refer
to a node, input, or output of a member component, use the syntax
comp_name.port_name, where comp_name is the identifier assigned
to the member component in the components declaration block and
port_name is the name of the node, input, or output in the member
component file.

The following syntax connects node a of the composite component to
node a of the member component c1, node b of the member component
c1 to node a of the member component c2, and node b of the member
component c2 to node b of the composite component.

```
connections
   connect(a, c1.a);
   connect(c1.b, c2.a);
   connect(c2.b, b);
end
```

See the connect reference page for more information on the connect
construct syntax.

Examples   This example implements a simple RC circuit that models the discharging of an initially charged capacitor. The composite component uses the components from the Simscape Foundation library as building blocks, and connects them as shown in the following block diagram.
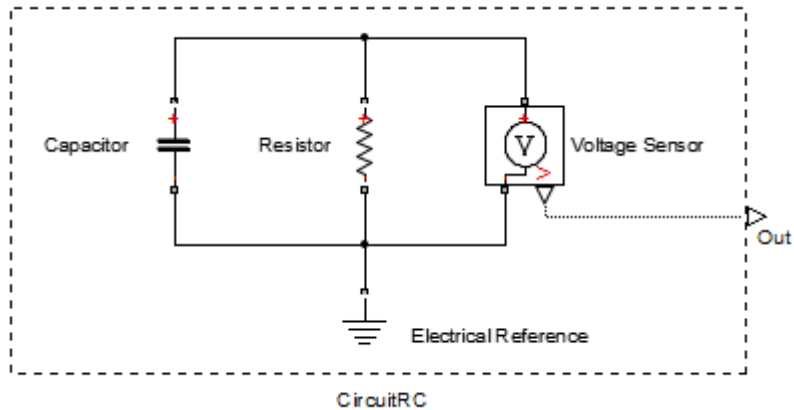
CircuitRC

```
component CircuitRC
   outputs
      Out = { 0.0, 'V' }; % I:right
   end
   parameters
      p1 = {1e-6, 'F'};  % Capacitance
      v = { 1, 'V' };    % Initial voltage
      p2 = {10, 'Ohm'};  % Resistance
   end
   components(Hidden=true)
      c1 = foundation.electrical.elements.capacitor(c=p1,v0=v);
      VoltSensor = foundation.electrical.sensors.voltage;
      r1 = foundation.electrical.elements.resistor(R=p2);
      Grnd = foundation.electrical.elements.reference;
   end
   connections
      connect(Grnd.V, c1.n, r1.n, VoltSensor.n);
```

```
        connect(VoltSensor.p, r1.p, c1.p);
        connect(VoltSensor.V, Out);
    end
end
```

The `connections` block contains three `connect` constructs:

- The first one connects the negative ports of the capacitor, resistor, and voltage sensor to each other and to ground

- The second one connects the positive ports of the capacitor, resistor, and voltage sensor to each other

- The third one connects the physical signal output port of the voltage sensor to the external output `Out` of the composite component

The resulting composite block has one physical signal output port, `Out`, and three externally adjustable parameters in the block dialog box: **Capacitance**, **Initial voltage**, and **Resistance**.

See Also

connect

"Declaring Member Components" on page 2-47

"Specifying Component Connections" on page 2-51

delay

Purpose Return past value of operand

Syntax
```
delay(u,tau)
delay(u,tau, History = uO, MaximumDelay = taumax)
```

Description Use the `delay` operator in the `equations` section to refer to past values of expressions:

`delay(u,tau) = u(t-tau)`

The full syntax is:

`delay(u,tau, History = uO, MaximumDelay = taumax)`

The required operands are:

- `u` — The first operand is the Simscape expression being delayed. It can be any numerical expression that does not itself include `delay` or `der` operators.

- `tau` — The second operand is the delay time. It must be a numerical expression with the unit of time. The value of `tau` can change, but it must remain strictly positive throughout the simulation.

The optional operands may appear in any order. They are:

- `History` — The return value for the initial time interval ($t <= StartTime + tau$). The units of `u` and `uO` must be commensurate. The default `uO` is 0.

- `MaximumDelay` — The maximum delay time. `taumax` must be a constant or parametric expression with the unit of time. If you specify `MaximumDelay = taumax`, a runtime error will be issued whenever `tau` becomes greater than `taumax`.

> **Note** You have to specify `MaximumDelay` if the delay time, `tau`,
> is not a constant or parametric expression. If `tau` is a constant
> or parametric expression, its value is used as the default for
> `MaximumDelay`, that is, `taumax = tau`.

At any time *t*, `delay(u,tau)` returns a value approximating *u*(*t -
tau*) for the current value of *tau*. More specifically, the expression
`delay(u,tau, History = u0)` is equivalent to

```
if t <= (StartTime + tau)
   return u0(t)
else
   return u(t-tau)
end
```

In other words, during the initial time interval, from the start of
simulation and until the specified delay time, `tau`, has elapsed, the
`delay` operator returns `u0` (or 0, if `History` is not specified). For
simulation times greater than `tau`, the `delay` operator returns the past
value of expression, *u*(*t - tau*).

Note

- When simulating a model that contains blocks with delays, memory
  allocation for storing the data history is controlled by the **Delay
  memory budget [kB]** parameter in the Solver Configuration block.
  If this budget is exceeded, simulation errors out. You can adjust this
  parameter value based on your available memory resources.

- For recommendation on how to linearize a model that contains blocks
  with delays, see "Linearizing with Simulink Linearization Blocks".

Examples

This example shows implementation for a simple dynamic system:

$$\dot{x} = -x(t-1)$$
$$x(t < 0) = 1$$

The Simscape file looks as follows:

```
component MyDelaySystem
  parameters
    tau = {1.0,'s'};
  end
  variables
    x = 1.0;
  end
  equations
    x.der == -delay( x,tau,History = 1.0 )*{ 1, '1/s' }; % x' = - x(t - 1)
  end
end
```

`MaximumDelay` is not required because `tau` is constant.

The `{ 1, '1/s' }` multiplication factor is used to reconcile the units of expression and its time derivative. See `der` reference page for more information.

For other examples of using the `delay` operator, see source for the PS Constant Delay and PS Variable Delay blocks in the Simscape Foundation library (open the block dialog box and click the **View source** link).

The Variable Transport Delay example shows how you can model a variable transport delay using the `delay` operator. To see the implementation details, open the example model, look under mask of the Transport Delay subsystem, then right-click the Variable Transport Delay block and select **View Simscape source**.

See Also

equations

Purpose   Return time derivative of operand

Syntax   der(x)
                x.der

Description The equations section may contain der operator, which returns the time derivative of its operand:

$$der(x) = x.der = \dot{x} = \frac{dx}{dt}$$

der operator takes any numerical expression as its argument:

- der applied to expressions that are continuous returns their time derivative

- der applied to time argument returns 1

- der applied to expressions that are parametric or constant returns 0

- der applied to countable operands returns 0. For example, der(a<b) returns 0 even if *a* and *b* are variables.

The return unit of der is the unit of its operand divided by seconds.

The following restrictions apply:

- You cannot form nonlinear expressions of the output from der. For example, der(x)*der(x) would produce an error because this is no longer a linearly implicit system.

- Higher order derivatives are not allowed. For example, der(der(x)) would produce an error.

- For a component to compile, the number of differential equations should equal the number of differential variables.

Examples   This example shows implementation for a simple dynamic system:

$$\dot{x} = 1 - x$$

The Simscape file looks as follows:

```
component MyDynamicSystem
  variables
    x = 0;
  end
  equations
    x.der == (1 - x)*{ 1, '1/s' };  % x' = 1 - x
  end
end
```

The reason you need to multiply by `{ 1, '1/s' }` is that `(1-x)` is unitless, while the left-hand side (`x.der`) has the units of 1/s. Both sides of the equation statement must have the same units.

See Also   equations

Purpose   Program customized run-time errors and warnings

Syntax   assert (*predicate_condition*, *message*, Warn = true|false);

Description   The equations section may contain the assert construct, which lets
you specify customized run-time errors and warnings:

assert (*predicate_condition*, *message*, Warn = true|false);

predicate_condition	The expression to be evaluated at run time. It can be a function of time, inputs, parameters, and variables.
message	Optional text string (with single quotes) that tells the block user why the run-time error or warning is triggered.
Warn = true|false	Optional attribute that specifies whether simulation errors out when the predicate condition is violated (Warn = false), or continues with a warning (Warn = true). The default is Warn = false.

You can use the assert construct in:

- The top-level equations.

- The if-elseif-else branches of a conditional expression.

- The expression clause and the right-hand side of the declaration clause of a let expression.

When you use an assert construct in a branch of a conditional
expression, it is not counted towards the number of expressions in the
branch, and is therefore exempt from the general rule that the total
number of equation expressions, their dimensionality, and their order
must be the same for every branch of the if-elseif-else statement.
For example, the following is valid:

```
if x>1
    y == 1;
else
    assert(b > 0);
    y == 3;
end
```

The scope of the `assert` construct is defined by the scope of its branch. In the preceding example, the predicate condition `b > 0` is evaluated only when the `else` branch is in effect, that is, when *x* is less than or equal to 1.

Examples   **Run-Time Error**

Generate a run-time error if the fluid volume in a reservoir becomes negative:

```
assert( V >= 0, 'Insufficient fluid volume for proper operation' );
```

During simulation, if the internal variable V (corresponding to the volume of fluid in the reservoir) assumes a negative value, simulation stops and outputs an error message containing the following information:

- Simulation time when the assertion got triggered

- The *message* string (in this example, `Insufficient fluid volume for proper operation`)

- An active link to the block that triggered the assertion. Click the `Block path` link to highlight the block in the model diagram.

- An active link to the assert location in the component source file. Click the `Assert location` link to open the Simscape source file of the component, with the cursor at the start of violated predicate condition. For Simscape protected files, the `Assert location` information is omitted from the error message.

Run-Time Warning

If you do not want simulation to stop, but still want to display a warning that a certain condition has been violated, set the Warn attribute to true. For example, if hydraulic pressure drops below fluid vapor saturation level at some point, this condition may result in cavitation and invalidate the modeling assumptions used in a block. You can add the following assert construct to the hydraulic component equations:

```
assert( p > p_cav, 'Pressure is below vapor level; cavitation possible
```

In this case, if the predicate condition is violated, the simulation continues, but the warning message appears in the MATLAB Command Window. The format of the warning message is the same as of the error message described in the previous example.

See Also equations

"Programming Run-Time Errors and Warnings" on page 2-43

domain

Purpose   Domain model keywords

Syntax
```
domain
variables
variables(Balancing = true)
parameters
```

Description   `domain` begins the domain model class definition, which is terminated by an `end` keyword. Only blank lines and comments can precede `domain`. You must place a domain model class definition in a file of the same name with a file name extension of `.ssc`.

`variables` begins an Across variables declaration block, which is terminated by an `end` keyword. 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.

`variables(Balancing = true)` begins a Through variables declaration block, which is terminated by an `end` keyword. 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 `through` block. This block is required.

Each variable is defined as a value with unit. See "Declaring Through and Across Variables for a Domain" on page 2-7 for more information.

`parameters` begins a domain parameters declaration block, which is terminated by an `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. This block is optional.

See "Propagation of Domain Parameters" on page 2-67 for more information.

Table of Attributes

For declaration member attributes, see "Attribute Lists" on page 2-76.

Examples   This 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
```

This file, named `t_hyd.ssc`, declares a hydraulic domain, with pressure
as an Across variable, flow rate as a Through variable, and an
associated domain parameter, fluid temperature.

```
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
```

See Also   component

equations

Purpose   Define component equations

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

Description `equations` begins the equation section in a component file; this
section is terminated by an `end` keyword. 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. All members declared in the component are available by
their name in the equation section.

The following syntax defines a simple equation.

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

The statement `Expression1 == Expression2` is an equation statement.
It specifies continuous mathematical equality between two objects of
class `Expression`. An `Expression` is any valid MATLAB expression
that does not use any of the relational operators: ==, <, >, <=, >=, ~=, &&,
||. `Expression` may be constructed from any of the identifiers defined
in the model declaration.

The equation section may contain multiple equation statements. You
can also specify conditional equations by using `if` statements as follows:

```
equations
if Expression
ExpressionList
{ elseif Expression
ExpressionList }
else
ExpressionList
end
```

```
end
```

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 define intermediate terms and use them in equations by using `let` statements as follows:

```
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.

---

equations

The following rules apply to the equation section:

- `EquationList` is one or more objects of class `EquationExpression`, separated by a comma, semicolon, or newline.

- `EquationExpression` can be one of:
  - `Expression`
  - Conditional expression (`if-elseif-else` statement)
  - Let expression (`let-in-end` statement)
- `Expression` is any valid MATLAB expression. It may be formed with the following operators:
  - Arithmetic
  - Relational (with restrictions, see "Use of Relational Operators in Equations" on page 2-24)
  - Logical
  - Primitive Math
  - Indexing
  - Concatenation
- In the equation section, `Expression` may not be formed with the following operators:
  - Matrix Inversion
  - MATLAB functions not listed in Supported Functions on page 4-29
- The `colon` operator may take only constants or `end` as its operands.
- All members of the component are accessible in the equation section, but none are writable.

The following MATLAB functions can be used in the equation section. The table contains additional restrictions that pertain only to the equation section. It also indicates whether a function is discontinuous.

If the function is discontinuous, it introduces a zero-crossing when used with one or more continuous operands.

Supported Functions

Name	Restrictions	Discontinuous
plus		
uplus		
minus		
uminus		
mtimes		
times		
mpower		
power		
mldivide	Nonmatrix denominator	
mrdivide	Nonmatrix denominator	
ldivide		
rdivide		
eq	Do not use with continuous variables	
ne	Do not use with continuous variables	
lt		
gt		
le		
ge		

Supported Functions (Continued)

Name	Restrictions	Discontinuous
and		Yes
or		Yes
sin		
cos		
tan		
asin		
acos		
atan		
atan2		
log		
log10		
sinh		
cosh		
tanh		
exp		
sqrt	For negative numbers, calculated as $\sqrt{\lvert x \rvert} \cdot sign(x)$. For example, sqrt(–1) = –1.	
abs		Yes
logical		Yes
sign		Yes
floor	Scalar argument	Yes

Supported Functions (Continued)

Name	Restrictions	Discontinuous
`ceil`	Scalar argument	Yes
`fix`	Scalar argument	Yes
`round`	Scalar argument	Yes

Examples

For a component where *x* and *y* are declared as 1x1 variables, specify an equation of the form $y = x^2$:

```
equations
  y == x^2;
end
```

For the same component, specify the following piecewise equation:

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

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

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

See Also

```
assert
```

```
delay
```

```
der
```

```
tablelookup
```

equations

time

Purpose   Define component inputs, that is, Physical Signal input ports of block

Syntax   inputs in1 = { value , 'unit' }; end

Description inputs begins a component inputs definition block, which is terminated
by an end keyword. This block contains declarations for component
inputs. Inputs will appear as Physical Signal input ports in the block
diagram when the component file is brought into a Simscape model.
Each input is defined as a value with unit, where value can be a scalar,
vector, or matrix. For a vector or a matrix, all signals have the same
unit.

Specifying an optional comment lets you control the port label and
location in the block icon.

The following syntax defines a component input, in1, as a value with
unit. value is the initial value. unit is a valid unit string, defined
in the unit registry.

```
inputs
    in1 = { value , 'unit' };
end
```

You can specify the input port label and location, the way you want it to
appear in the block diagram, as a comment:

```
inputs
    in1 = { value , 'unit' };  % label:location
end
```

where label is a string corresponding to the input port name in the
block diagram, location is one of the following strings: left, right,
top, bottom.

Examples   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
```

The next example declares an input port I as a row vector of electrical currents. The three signals have a default value of 1 A. The signal initial values may be different, but the unit has to be the same.

```
 inputs
   I = { [1 1 1], 'A'};
 end
```

See Also   nodes

outputs

Purpose   Define component nodes, that is, conserving ports of block

Syntax   `nodes a = package_name.domain_name; end`

Description   `nodes` begins a nodes declaration block, which is terminated by an `end` keyword. This block contains declarations for all the component nodes, which correspond to the conserving ports of a Simscape block generated from the component file. Each node is defined by assignment to an existing domain. See "Declaring Component Nodes" on page 2-11 for more information.

The following syntax defines a node, `a`, by associating it with a domain, `domain_name`. `package_name` is the full path to the domain, starting with the top package directory. For more information on packaging your Simscape files, see "Generate Custom Block Libraries from Simscape Component Files" on page 3-2.

```
nodes
    a = package_name.domain_name;
end
```

You can specify the port label and location, the way you want it to appear in the block diagram, as a comment:

```
nodes
    a = package_name.domain_name;  % label:location
end
```

where `label` is a string corresponding to the port name in the block diagram, `location` is one of the following strings: `left`, `right`, `top`, `bottom`.

Examples   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
```

The following example declares an electrical node using the syntax for the Simscape Foundation electrical domain. In the block diagram, this port will be labelled + and will be located on the top side of the block icon.

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

See Also
inputs

outputs

Purpose   Define component outputs, that is, Physical Signal output ports of block

Syntax   outputs out1 = { value , 'unit' }; end

Description   outputs begins a component outputs definition block, which is
                   terminated by an end keyword. This block contains declarations for
                   component outputs. Outputs will appear as Physical Signal output
                   ports in the block diagram when the component file is brought into a
                   Simscape model. Each output is defined as a value with unit, where
                   value can be a scalar, vector, or matrix. For a vector or a matrix, all
                   signals have the same unit.

                   Specifying an optional comment lets you control the port label and
                   location in the block icon.

                   The following syntax defines a component output, out1, as a value with
                   unit. value is the initial value. unit is a valid unit string, defined
                   in the unit registry.

```
outputs
    out1 = { value , 'unit' };
end
```

                   You can specify the output port label and location, the way you want it
                   to appear in the block diagram, as a comment:

```
outputs
    out1 = { value , 'unit' };  % label:location
end
```

                   where label is a string corresponding to the input port name in the
                   block diagram, location is one of the following strings: left, right,
                   top, bottom.

Examples   The following example declares an output port p, with a default value of
                   1 Pa, specifying the output port of a hydraulic pressure sensor. In the
                   block diagram, this port will be named **Pressure** and will be located on
                   the bottom side of the block icon.

outputs

```
outputs
    p = { 1 'Pa' };    % Pressure:bottom
end
```

The next example declares an output port v as a 3-by-3 matrix of linear velocities.

```
outputs
  v = {zeros(3), 'm/s'};
end
```

See Also   inputs

nodes

Purpose   Specify component parameters

Syntax   `parameters comp_par1 = { value , 'unit' };  end`

Description   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.

`parameters` begins a component parameters definition block, which is terminated by an `end` keyword. This block contains declarations for component parameters. Parameters will appear in the block dialog box when the component file is brought into a Simscape model. Each parameter is defined as a value with unit. Specifying an optional comment lets you control the parameter name in the block dialog box.

The following syntax defines a component parameter, `comp_par1`, as a value with unit. `value` is the initial value. `unit` is a valid unit string, defined in the unit registry.

```
parameters
    comp_par1 = { value , 'unit' };
end
```

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 array `{ value , '1' }`.

You can specify the parameter name, the way you want it to appear in the block dialog box, as a comment:

```
parameters
```

parameters

```
        comp_par1 = { value , 'unit' }; % Parameter name
    end
```

Examples   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
```

See Also   variables

Purpose   Prepare component for simulation

Syntax   function setup
                   [...]
                  end

Description   function setup
                  [...]
                  end

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.

Use the setup function for the following purposes:

- Validating parameters

- Computing derived parameters

- Setting initial conditions

- Relating inputs, outputs, and variables 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.

- 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 "Persistent Variables" and "Global 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).

Examples   The following setup function checks the value of a parameter MyParam, declared in the declaration section of a component file. It defines a maximum allowed value for this parameter, MaxValue, and if MyParam is greater than MaxValue, overrides it with MaxValue and issues a warning.

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

See Also   across

through

Purpose   Return value based on interpolating set of data points

Syntax   ```
tablelookup(x1d, x2d, yd, x1, x2,
    interpolation = linear|cubic|spline,
    extrapolation = linear|nearest)
```

**Description**  Use the `tablelookup` function in the `equations` section to compute
an output value by interpolating the input value against a set of
data points. This functionality is similar to that of the Simulink and
Simscape Lookup Table blocks. It allows you to incorporate table-driven
modeling directly in your custom block, without the need of connecting
an external Lookup Table block to your model.

The `tablelookup` function supports one-dimensional and
two-dimensional lookup tables. The full syntax is:

```
tablelookup(x1d, x2d, yd, x1, x2, interpolation =
linear|cubic|spline, extrapolation = linear|nearest)
```

`x1d`	Data set of input values along the first direction, specified as a one-dimensional array. The values must be strictly monotonic, either increasing or decreasing. This is a required argument.
`x2d`	Data set of input values along the second direction, specified as a one-dimensional array. The values must be strictly monotonic, either increasing or decreasing. This argument is used only for the two-dimensional table lookup.

yd	Data set of output values for the table lookup. This is a required argument. For one-dimensional table lookup, yd must be a one-dimensional array of the same size as x1d.
	For two-dimensional table lookup, yd must be a matrix, with the size matching the dimensions defined by the input data sets. For example, if x1d is a 1-by-m array, and x2d is a 1-by-n array, then yd must be an m-by-n matrix.
x1	The input value along the first direction. Its units must be commensurate with the units of x1d. This is a required argument.
x2	The input value along the second direction. Its units must be commensurate with the units of x2d. This argument is used only for the two-dimensional table lookup.
interpolation = linear|cubic|spline	Optional argument that specifies the approximation method for calculating the output value when the input value is inside the range specified in the lookup table. The default is interpolation = linear.
extrapolation = linear|nearest	Optional argument that specifies the approximation method for calculating the output value when the input value is outside the range specified in the lookup table. The default is extrapolation = linear.

The interpolation argument values are:

- linear — For one-dimensional table lookup, uses a linear function. For two-dimensional table lookup, uses a bilinear interpolation

algorithm, which is an extension of linear interpolation for functions in two variables. The method performs linear interpolation first in *x*-direction and then in *y*-direction.

- cubic — For one-dimensional table lookup, uses the Piecewise Cubic Hermite Interpolation Polinomial (PCHIP). For more information, see the pchip MATLAB function. For two-dimensional table lookup, uses the bicubic interpolation algorithm described in [1].

- spline — For one-dimensional table lookup, uses the cubic spline interpolation algorithm described in [1]. For two-dimensional table lookup, uses the bicubic spline interpolation algorithm described in [1].

The extrapolation argument values are:

- linear— Extrapolates using the linear method, based on the last two output values at the appropriate end of the range. That is, the function uses the first and second specified output values if the input value is below the specified range, and the two last specified output values if the input value is above the specified range.

- nearest — Uses the last specified output value at the appropriate end of the range. That is, the function uses the last specified output value for all input values greater than the last specified input argument, and the first specified output value for all input values less than the first specified input argument.

The function returns an output value, in the units specified for yd, by looking up or estimating table values based on the input values:

When inputs x1 and x2...	The tablelookup function...
Match the values of indices in the input data sets, x1d and x2d	Outputs the corresponding table value, yd
Do not match the values of indices in the input data sets, but are within range	Interpolates appropriate table values, using the method specified as the interpolation argument value
Do not match the values of indices in the input data sets, and are out of range	Extrapolates the output value, using the method specified as the extrapolation argument value

Error Checking

The following rules apply to data sets x1d, x2d, and yd:

- For one-dimensional table lookup, x1d and yd must be one-dimensional arrays of the same size.

- For two-dimensional table lookup, x1d and x2d must be one-dimensional arrays, and yd must be a matrix, with the size matching the dimensions defined by the input data sets. For example, if x1d is a 1-by-m array, and x2d is a 1-by-n array, then yd must be an m-by-n matrix.

- The x1d and x2d values must be strictly monotonic, either increasing or decreasing.

- For cubic or spline interpolation, each data set must contain at least three values. For linear interpolation, two values are sufficient.

If these rules are violated by...	You get an error...
The block author, in the component Simscape file	At build time, when running `ssc_build` to convert the component to a custom block
The block user, when entering values in the block dialog box	At simulation time, when attempting to simulate the model containing the custom block

Examples

1D Lookup Table Implementation

This example implements a one-dimensional lookup table with linear interpolation and extrapolation.

```
component tlu_1d_linear
 inputs
   u = 0;
 end
 outputs
   y = 0;
 end
 parameters (Size=variable)
   xd = [1 2 3 4];
   yd = [1 2 3 4];
 end
 equations
   y == tablelookup(xd, yd, u);
 end
end
```

`xd` and `yd` are declared as variable-size parameters. This enables the block users to provide their own data sets when the component is converted to a custom block. For more information, see "Using Lookup Tables in Equations" on page 2-40.

The `xd` values must be strictly monotonic, either increasing or decreasing. `yd` must have the same size as `xd`.

2D Lookup Table Implementation

This example implements a two-dimensional lookup table with specific interpolation and extrapolation methods.

```
component tlu_2d
 inputs
   u1 = 0;
   u2 = 0;
 end
 outputs
   y = 0;
 end
 parameters (Size=variable)
   x1d = [1 2 3 4];
   x2d = [1 2 3];
   yd = [1 2 3; 3 4 5; 5 6 7; 7 8 9];
 end
 equations
   y == tablelookup(x1d, x2d, yd, u1, u2, interpolation=spline, extrapola
 end
end
```

x1d, x2d, and yd are declared as variable-size parameters. The x1d and x2d vector values must be strictly monotonic, either increasing or decreasing. For spline interpolation, each vector must have at least three values. The size of the yd matrix must match the dimensions of the x1d and x2d vectors.

The interpolation uses the bicubic spline algorithm. The extrapolation uses the nearest value of yd for out-of-range u1 and u2 values.

Using Lookup Table with Units

This example implements a one-dimensional lookup table with units, to map temperature to pressure, with linear interpolation and extrapolation.

```
component TtoP
 inputs
```

```
  u = {0, 'K'}; % temperature
 end
 outputs
  y = {0, 'Pa'}; % pressure
 end
 parameters (Size=variable)
  xd = {[100 200 300 400] 'K'};
  yd = {[1e5 2e5 3e5 4e5] 'Pa'};
 end
 equations
  y == tablelookup(xd, yd, u);
 end
end
```

xd and yd are declared as variable-size parameters with units.
This enables the block users to provide their own data sets when
the component is converted to a custom block, and also to select
commensurate units from the drop-downs in the custom block dialog
box. For more information, see "Using Lookup Tables in Equations"
on page 2-40.

The xd values must be strictly monotonic, either increasing or
decreasing. yd must have the same size as xd.

References   [1] W.H.Press, B.P.Flannery, S.A.Teulkolsky, W.T.Wetterling,
Numerical Recipes in C: The Art of Scientific Computing, Cambridge
University Press, 1992

See Also   equations

through

Purpose Establish relationship between component variables and nodes

Syntax through(variableI, node1.variableA, node2.variableB)

Description through(variableI, node1.variableA, node2.variableB)
establishes the following relationship between the three arguments:
for each variableI, node1.variableA is assigned the value sum(
variableI) and node2.variableB is assigned the value sum(
-variableI). All arguments are variables. The first one is not
associated with a node. The second and third must be associated with
a node.

The following rules apply:

- All arguments must have consistent units.
- The second and third arguments do not need to be associated with the
  same domain. For example, one may be associated with a one-phase
  electrical domain, and the other with a 3-phase electrical.
- Either the second or the third argument may be replaced with [ ]
  to indicate the reference node.

Examples For example, if a component declaration section contains two electrical
nodes, p and n, and a variable i = { 0, 'A' }; specifying current,
you can establish the following relationship in the setup section:

through(i, p.i, n.i);

This defines current i as a Through variable from node p to node n.

See Also across

Purpose   Access global simulation time

Syntax   `time`

Description   You can access global simulation time from the equation section of a Simscape file using the `time` function.

time returns the simulation time in seconds.

Examples   The following example illustrates $y = \sin(\omega t)$:

```
component
  parameters
    w = { 1, `1/s' } % omega
  end
  outputs
    y = 0;
  end
  equations
    y == sin( w * time );
  end
end
```

See Also   `equations`

value

Purpose	Convert variable or parameter to unitless value with specified unit conversion
Syntax	`value(a,'unit')`
	`value(a,'unit','type')`
Description	`value(a,'unit')` returns a unitless numerical value, converting `a` into units `unit`. `a` is a variable or parameter, specified as a value with unit, and `unit` is a unit defined in the unit registry. `unit` must be commensurate with the units of `a`.
	`value(a,'unit','type')` performs either linear or affine conversion of temperature units and returns a unitless numerical value, converting `a` into units `unit`. `type` specifies the conversion type and can be one of two strings: `linear` or `affine`. If the type is not specified when converting temperature units, it is assumed to be affine.
	Use this function in the setup and equation sections of a Simscape file to convert a variable or parameter into a scalar value.
Examples	If `a = { 10, `cm' }`, then `value(a, 'm')` returns 0.1.
	If `a = { 10, `C' }`, then `value(a, 'K', 'linear')` returns 10.
	If `a = { 10, `C' }`, then `value(a, 'K', 'affine')` returns 283.15. `value(a, 'K')` also returns 283.15.
	If `a = { 10, `cm' }`, then `value(a, 's')` issues an error because the units are not commensurate.

Purpose   Define domain or component variables

Syntax   variables comp_var1 = { value , 'unit' }; end

variables domain_across_var1 = { value , 'unit' }; end

variables(Balancing = true) domain_through_var1 = { value , 'unit' };

Description variables begins a variables declaration block, which is terminated by
an end keyword. In a component file, this block contains declarations
for all the variables associated with the component. In a domain file,
this block contains declarations for all the Across variables associated
with the domain. Additionally, domain files must have a separate
variables declaration block, with the Balancing attribute set to true,
which contains declarations for all the Through variables associated
with the domain.

In a component file, the following syntax defines an Across, Through, or
internal variable, comp_var1, as a value with unit. value is the initial
value. unit is a valid unit string, defined in the unit registry.

```
variables
    comp_var1 = { value , 'unit' };
end
```

In a domain file, the following syntax defines an Across variable,
domain_across1, as a value with unit. value is the initial value. unit
is a valid unit string, defined in the unit registry.

```
variables
    domain_across_var1 = { value , 'unit' };
end
```

variables

In a domain file, the following syntax defines a Through variable, `domain_through1`, as a value with unit. `value` is the initial value. `unit` is a valid unit string, defined in the unit registry.

```
variables(Balancing = true)
    domain_through_var1 = { value , 'unit' };
end
```

Examples   The following example initializes the variable w (angular velocity) as 0 rad/s:

```
variables
    w = { 0, 'rad/s' };
end
```

The following example initializes the domain Through variable t (torque) as 1 N*m:

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

See Also   "Declaring Component Variables" on page 2-8

"Declaring Through and Across Variables for a Domain" on page 2-7

5

Simscape Foundation Domains

Foundation Domain Types and Directory Structure

Simscape software comes with the following Foundation domains:

- "Electrical Domain" on page 5-4
- "Hydraulic Domain" on page 5-5
- "Magnetic Domain" on page 5-7
- "Mechanical Rotational Domain" on page 5-8
- "Mechanical Translational Domain" on page 5-9
- "Pneumatic Domain" on page 5-10
- "Thermal Domain" on page 5-12

Simscape Foundation libraries are organized in a package containing domain and component Simscape files. The name of the top-level package directory is +foundation, and the package consists of subpackages containing domain files, structured as follows:

```
- +foundation
|-- +electrical
| |-- electrical.ssc
| |-- ...
|-- +hydraulic
| |-- hydraulic.ssc
| |-- ...
|-- +magnetic
| |-- magnetic.ssc
| |-- ...
|-- +mechanical
| |-- +rotational
| | |-- rotational.ssc
| | |-- ...
| |-- +translational
| | |-- translational.ssc
| | |-- ...
|-- +pneumatic
| |-- pneumatic.ssc
| |-- ...
```

```
|-- +thermal
|  |-- thermal.ssc
|  |-- ...
```

To use a Foundation domain in a component declaration, refer to the domain name using the full path, starting with the top package directory. The following example uses the syntax for the Simscape Foundation mechanical rotational domain:

```
r = foundation.mechanical.rotational.rotational;
```

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.

Electrical Domain

The electrical domain declaration is shown below.

```
domain electrical
% Electrical Domain

% Copyright 2005-2008 The MathWorks, Inc.

  parameters
    Temperature = { 300.15 , 'K'     }; % Circuit temperature
    GMIN        = { 1e-12  , '1/Ohm' }; % Minimum conductance, GMIN
  end

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

  variables(Balancing = true)
    i = { 0 , 'A' };
  end

end
```

It contains the following variables and parameters:

- Across variable v (voltage), in volts

- Through variable i (current), in amperes

- Parameter *Temperature*, specifying the circuit temperature

- Parameter *GMIN*, specifying minimum conductance

To refer to this domain in your custom component declarations, use the following syntax:

```
foundation.electrical.electrical
```

Hydraulic Domain

The hydraulic domain declaration is shown below.

```
domain hydraulic
% Hydraulic Domain

% Copyright 2005-2008 The MathWorks, Inc.

  parameters
    density      = { 850   , 'kg/m^3' }; % Fluid density
    viscosity_kin = { 18e-6 , 'm^2/s'  }; % Kinematic viscosity
    bulk         = { 0.8e9 , 'Pa'     }; % Bulk modulus at atm. pressure and no gas
    alpha        = { 0.005 , '1'      }; % Relative amount of trapped air
  end

  variables
    p = { 0 , 'Pa' };
  end

  variables(Balancing = true)
    q = { 0 , 'm^3/s' };
  end

end
```

It contains the following variables and parameters:

- Across variable *p* (pressure), in Pa

- Through variable *q* (flow rate), in m^3/s

- Parameter *density*, specifying the default fluid density

- Parameter *viscosity_kin*, specifying the default kinematic viscosity

- Parameter *bulk*, specifying the default fluid bulk modulus at atmospheric pressure and no gas

- Parameter *alpha*, specifying the default relative amount of trapped air in the fluid

To refer to this domain in your custom component declarations, use the following syntax:

```
foundation.hydraulic.hydraulic
```

Magnetic Domain

The magnetic domain declaration is shown below.

```
domain magnetic
% Magnetic Domain

% Copyright 2009 The MathWorks, Inc.

  parameters
    mu0 = { 4*pi*1e-7 'Wb/(m*A)' };   % Permeability constant
  end

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

  variables(Balancing = true)
    phi = { 0 , 'Wb' };
  end

end
```

It contains the following variables and parameters:

- Across variable *mmf* (magnetomotive force), in A
- Through variable *phi* (flux), in Wb
- Parameter *mu0*, specifying the permeability constant of the material

To refer to this domain in your custom component declarations, use the following syntax:

```
foundation.magnetic.magnetic
```

Mechanical Rotational Domain

The mechanical rotational domain declaration is shown below.

```
domain rotational
% Mechanical Rotational Domain

% Copyright 2005-2008 The MathWorks, Inc.

  variables
    w = { 0 , 'rad/s' };
  end

  variables(Balancing = true)
    t = { 0 , 'N*m' };
  end

end
```

It contains the following variables:

- Across variable *w* (angular velocity), in rad/s

- Through variable *t* (torque), in N*m

To refer to this domain in your custom component declarations, use the following syntax:

```
foundation.mechanical.rotational.rotational
```

Mechanical Translational Domain

The mechanical translational domain declaration is shown below.

```
domain translational
% Mechanical Translational Domain

% Copyright 2005-2008 The MathWorks, Inc.

  variables
    v = { 0 , 'm/s' };
  end

  variables(Balancing = true)
    f = { 0 , 'N' };
  end

end
```

It contains the following variables:

- Across variable v (velocity), in m/s

- Through variable f (force), in N

To refer to this domain in your custom component declarations, use the following syntax:

```
foundation.mechanical.translational.translational
```

Pneumatic Domain

The pneumatic domain declaration is shown below.

```
domain pneumatic
% Pneumatic 1-D Flow Domain

% Copyright 2008-2009 The MathWorks, Inc.

  parameters
    gam = { 1.4, '1' };                 % Ratio of specific heats
    c_p = { 1005 , 'J/kg/K' };          % Specific heat at constant pressure
    c_v = { 717.86 , 'J/kg/K' };        % Specific heat at constant volume
    R   = { 287.05, 'J/kg/K' };         % Specific gas constant
    viscosity = { 18.21e-6, 'Pa*s' };   % Viscosity
    Pa  = { 101325, 'Pa' };             % Ambient pressure
    Ta  = { 293.15, 'K' };              % Ambient temperature
  end

  variables
    p = { 0 , 'Pa' };
    T = { 0 , 'K' };
  end

  variables(Balancing = true)
    G = { 0 , 'kg/s' };
    Q = { 0 , 'J/s' };
  end

end
```

It contains the following variables and parameters:

- Across variable p (pressure), in Pa

- Through variable G (mass flow rate), in kg/s

- Across variable T (temperature), in kelvin

- Through variable Q (heat flow), in J/s

- Parameter *gam*, defining the ratio of specific heats

- Parameter *c_p*, defining specific heat at constant pressure

- Parameter *c_v*, defining specific heat at constant volume

- Parameter *R*, defining specific gas constant

- Parameter *viscosity*, specifying the gas viscosity

- Parameter *Pa*, specifying the ambient pressure

- Parameter *Ta*, specifying the ambient temperature

These parameter values correspond to gas properties for dry air and ambient conditions of 101325 Pa and 20 degrees Celsius.

To refer to this domain in your custom component declarations, use the following syntax:

```
foundation.pneumatic.pneumatic
```

Thermal Domain

The thermal domain declaration is shown below.

```
domain thermal
% Thermal domain

% Copyright 2005-2008 The MathWorks, Inc.

  variables
    T = { 0 , 'K' };
  end

  variables(Balancing = true)
    Q = { 0 , 'J/s' };
  end

end
```

It contains the following variables:

- Across variable T (temperature), in kelvin

- Through variable Q (heat flow), in J/s

To refer to this domain in your custom component declarations, use the following syntax:

```
foundation.thermal.thermal
```

Index