Simscape™

Language Guide

MATLAB® & SIMULINK®

R2020**b**

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

Simscape™ Language Guide

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)
September 2013	Online only	Revised for Version 3.10 (Release 2013b)
March 2014	Online only	Revised for Version 3.11 (Release 2014a)
October 2014	Online only	Revised for Version 3.12 (Release 2014b)
March 2015	Online only	Revised for Version 3.13 (Release 2015a)
September 2015	Online only	Revised for Version 3.14 (Release 2015b)
October 2015	Online only	Rereleased for Version 3.13.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 4.0 (Release 2016a)
September 2016	Online only	Revised for Version 4.1 (Release 2016b)
March 2017	Online only	Revised for Version 4.2 (Release 2017a)
September 2017	Online only	Revised for Version 4.3 (Release 2017b)
March 2018	Online only	Revised for Version 4.4 (Release 2018a)
September 2018	Online only	Revised for Version 4.5 (Release 2018b)
March 2019	Online only	Revised for Version 4.6 (Release 2019a)
September 2019	Online only	Revised for Version 4.7 (Release 2019b)
March 2020	Online only	Revised for Version 4.8 (Release 2020a)
September 2020	Online only	Revised for Version 5.0 (Release 2020b)

Contents

1 Simscape Language Fundamentals

2 Creating Custom Components and Domains

4 Simscape File Deployment

Simscape Language Fundamentals

What Is the Simscape Language?

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

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

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

See Also

Related Examples

- "Model Linear Resistor in Simscape Language" on page 1-3

More About

- "Typical Simscape Language Tasks" on page 1-6
- "Simscape File Types and Structure" on page 1-8
- "Creating Custom Components" on page 1-13
- "When to Define a New Physical Domain" on page 1-11

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' };     % Current
    v = { 0, 'V' };     % Voltage
  end
  parameters
    R = { 1, 'Ohm' };   % Resistance
  end

  branches
    i : p.i -> n.i;
  end

  equations
    assert(R>0)
    v == p.v - n.v;
    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. Comments start with the `%` character.

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 Through and Across variables later in the file. You connect the component and domain variables by specifying the connection between the component variables and nodes.

All the public component variables appear on the **Variables** tab of the dialog box of the block generated from the component file. To specify how the name of the variable appears in the dialog box, use the comment immediately following the variable declaration (`Current` and `Voltage`).

- 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.

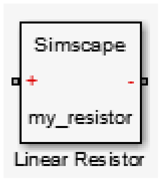
The `branches` section establishes the relationship between the component Through variable and the component nodes (and therefore the domain Through variable). The `i : p.i -> n.i` statement indicates that the current through the resistor flows from node `p` to node `n`.

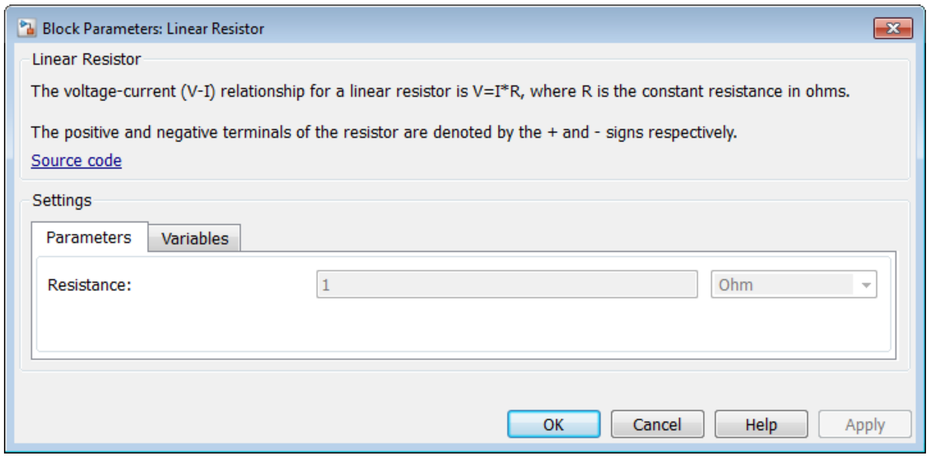
The final section contains the equations:

- The `assert` construct performs parameter validation, by checking that the resistance value is greater than zero. If the block parameter is set incorrectly, the `assert` triggers a run-time error.
- The first equation, `v == p.v - n.v`, establishes the relationship between the component Across variable and the component nodes (and therefore the domain Across variable). It defines the voltage across the resistor as the difference between the node voltages.
- The second equation, `v == i*R`, describes the operation of a linear resistor based on Ohm's law. 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 these equations 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

The voltage-current (V-I) relationship for a linear resistor is V=I*R, where R is the constant resistance in ohms.

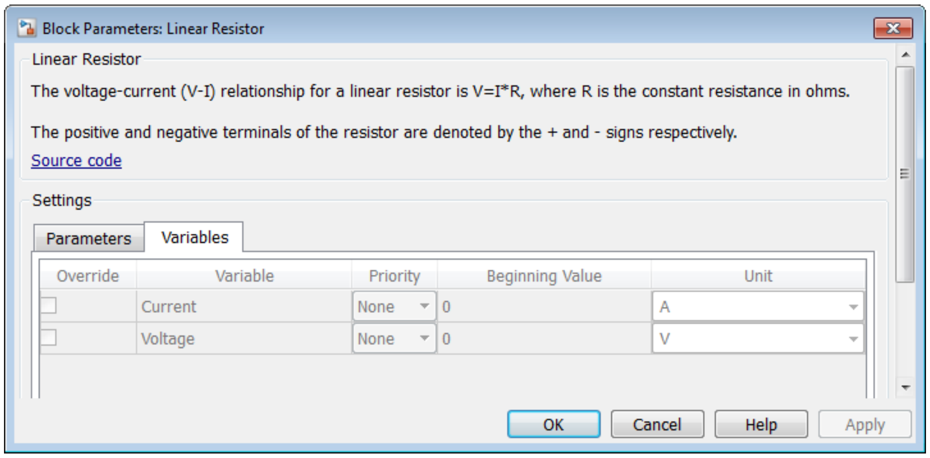
The positive and negative terminals of the resistor are denoted by the + and - signs respectively.

Source code

Settings

Parameters | Variables

Resistance: 1 Ohm



Block Parameters: Linear 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.

Source code

Settings

Parameters | Variables

Override	Variable	Priority	Beginning Value	Unit
☐	Current	None	0	A
☐	Voltage	None	0	V

To learn more about writing Simscape files and converting your textual components into custom Simscape blocks, refer to the following table.

For...	See...
Declaration semantics, rules, and examples	“Declaring Domains and Components” on page 2-3
Detailed information on writing component equations	“Defining Component Equations” on page 2-24
Annotating the component file to improve the generated block cosmetics and usability	“Customizing the Block Name and Appearance” on page 4-33
Generating Simscape blocks from component files	“Generating Custom Blocks from Simscape Component Files” on page 4-2

See Also

Related Examples

- “Mechanical Component — Spring” on page 2-90
- “Electrical Component — Ideal Capacitor” on page 2-91
- “No-Flow Component — Voltage Sensor” on page 2-92
- “Grounding Component — Electrical Reference” on page 2-93
- “Composite Component — DC Motor” on page 2-95

More About

- “What Is the Simscape Language?” on page 1-2
- “Simscape File Types and Structure” on page 1-8
- “Creating Custom Components” on page 1-13
- “When to Define a New Physical Domain” on page 1-11

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 1-13 “Declaring Domains and Components” on page 2-3 “Defining Component Equations” on page 2-24	“Declare a Spring Component” on page 2-19 “Mechanical Component — Spring” on page 2-90 “Electrical Component — Ideal Capacitor” on page 2-91 “No-Flow Component — Voltage Sensor” on page 2-92 “Grounding Component — Electrical Reference” on page 2-93
Create a custom component model constructed of other components	“About Composite Components” on page 2-58 “Declaring Member Components” on page 2-59 “Parameterizing Composite Components” on page 2-60 “Specifying Component Connections” on page 2-64	“Composite Component — DC Motor” on page 2-95
Generate a custom block from a Simscape component file	“Selecting Component File Directly from Block” on page 4-3 “Customizing the Block Name and Appearance” on page 4-33	“Deploy a Component File in Block Diagram” on page 4-5 “Customize Block Display” on page 4-45

Task	Background Information	Examples
Add a custom block library to Simscape libraries	“Building Custom Block Libraries” on page 4-25 “Using Source Protection for Simscape Files” on page 4-26 “Customizing the Library Name and Appearance” on page 4-29 “Customizing the Block Name and Appearance” on page 4-33	“Create a Custom Block Library” on page 4-31 “Customize Block Display” on page 4-45
Define a new domain, with associated Through and Across variables, and then use it in custom components	“When to Define a New Physical Domain” on page 1-11 “Declaring Domains and Components” on page 2-3	“Declare a Mechanical Rotational Domain” on page 2-18 “Propagation of Domain Parameters” on page 2-98
Create a component that supplies domain-wide parameters (such as fluid temperature) to the rest of the model	“Working with Domain Parameters” on page 2-98	“Custom Library with Propagation of Domain Parameters” on page 2-100

Simscape File Types and Structure

In this section...
"Simscape File Type" on page 1-8
"File and Model Types" on page 1-8
"Model File Structure" on page 1-9

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 4-25.

File and Model Types

There are two types of Simscape files that correspond 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 area 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 simple thermohydraulic orifice, you can create a domain model first, `t_hyd.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. For an example, see "Custom Library with Propagation of Domain Parameters" on page 2-100.

The third file type, function files, represents Simscape functions. Simscape functions model a class of pure first-order mathematical functions with explicit input-output relationship. Their purpose is to reuse expressions in equations and member declarations of multiple components.

Model 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
```

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 the following sections:
  - Structure — For composite components, describes how the constituent components' ports are connected to one another and to the external inputs, outputs, and nodes of the top-level component. Executed once for each instance of the component in the top-level model during model compilation.
  - Equation — For behavioral components, describes underlying equations. Executed throughout simulation.
  - Events — For discrete event modeling, lets you perform discrete changes on continuous variables. 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.

See Also

Related Examples

- “Model Linear Resistor in Simscape Language” on page 1-3

More About

- “What Is the Simscape Language?” on page 1-2

- “Typical Simscape Language Tasks” on page 1-6
- “Creating Custom Components” on page 1-13
- “When to Define a New Physical Domain” on page 1-11

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, Simscape Fluids™ or Simscape Electrical™ 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 6-2.

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 simple thermal hydraulic 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 2-98).

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.

See Also

More About

- “Foundation Domain Types and Directory Structure” on page 6-2
- “How to Define a New Physical Domain” on page 1-12
- “Declare a Mechanical Rotational Domain” on page 2-18

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 “Declare Through and Across Variables for a Domain” on page 2-6.

The parameters declaration block is optional. If present, it must begin with the `parameters` keyword and be terminated by the `end` keyword. This block contains declarations for domain parameters. These parameters are associated with the domain and can be propagated through the network to all components connected to the domain. For more information, see “Working with Domain Parameters” on page 2-98.

For an example of a domain file, see “Declare a Mechanical Rotational Domain” on page 2-18.

See Also

Related Examples

- “Declare a Mechanical Rotational Domain” on page 2-18
- “Declare Through and Across Variables for a Domain” on page 2-6

More About

- “When to Define a New Physical Domain” on page 1-11
- “Working with Domain Parameters” on page 2-98

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 Simscape Fluids Isothermal 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 "How to Define a New Physical Domain" on page 1-12.

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 may contain the following sections, appearing in any order:

- 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 2-3.
- Branches — Establishes the relationship between the component variables and nodes. This relationship connects the Through and Across variables declared inside the component to the domain Through and Across variables For more information, see "Define Relationship Between Component Variables and Nodes" on page 2-21.
- Structure — Declares the component connections for composite models. For more information, see "Specifying Component Connections" on page 2-64.

- 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 2-24.
- Events — Manages the event updates. Event modeling lets you perform discrete changes on continuous variables. For more information, see “Discrete Event Modeling” on page 2-52.
- Annotations — Lets you provide annotations in a component file that control various cosmetic aspects of a Simscape block generated from this component. See `annotations` for more information.

Generating a Custom Block from a Component File

After you have created a textual component file, you can deploy it directly into a block diagram using the workflows described in “Selecting Component File Directly from Block” on page 4-3. You can control the block name and appearance by using optional comments in the component file. For more information, see “Customizing the Block Name and Appearance” on page 4-33.

Adding a Custom Block Library

Adding a custom block library involves creating new components that model the desired physical behavior and structure. 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 “Building Custom Block Libraries” on page 4-25. 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 4-33.

See Also

Related Examples

- “Mechanical Component — Spring” on page 2-90
- “Electrical Component — Ideal Capacitor” on page 2-91
- “No-Flow Component — Voltage Sensor” on page 2-92
- “Grounding Component — Electrical Reference” on page 2-93
- “Composite Component — DC Motor” on page 2-95

More About

- “What Is the Simscape Language?” on page 1-2
- “Typical Simscape Language Tasks” on page 1-6
- “Declaring Domains and Components” on page 2-3
- “Defining Component Equations” on page 2-24
- “About Composite Components” on page 2-58
- “Building Custom Block Libraries” on page 4-25

Creating Custom Components and Domains

Declaring Domains and Components

In this section...

Declaration Section Purpose

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 other sections of a component file (such as branches, equations, and so on), 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 2-59. • 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 on page 2-5	Default value	Yes
`variables`	`domain` `component`	Numerical value with unit on page 2-5	Nominal value and default initial condition	Yes
`inputs`	`component`	Scalar, vector, or matrix double value with unit on page 2-5, or untyped	Default value, if typed	No
`outputs`	`component`	Scalar, vector, or matrix double value with unit on page 2-5, or untyped	Default value, if typed	No
`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 2-59)	No

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' }`.

See Also

Related Examples

- “Declare a Spring Component” on page 2-19
- “Declare a Mechanical Rotational Domain” on page 2-18
- “Declare Through and Across Variables for a Domain” on page 2-6
- “Declare Component Variables” on page 2-7
- “Declare Component Parameters” on page 2-11
- “Declaring Domain Parameters” on page 2-98
- “Declare Component Nodes” on page 2-14
- “Declare Component Inputs and Outputs” on page 2-16

Declare 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”.

`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 on page 2-5:

```
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 2-18 for more information.

See Also

Related Examples

- “Declare a Mechanical Rotational Domain” on page 2-18
- “Declare Component Variables” on page 2-7
- “Declare Component Nodes” on page 2-14
- “Declaring Domain Parameters” on page 2-98

More About

- “Declaring Domains and Components” on page 2-3

Declare Component Variables

In this section...

Through and Across 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 on page 2-5 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 Through and Across variables, you have to specify their relationship with component nodes, and therefore with the domain Through and Across variables. For more information, see "Define Relationship Between Component Variables and Nodes" on page 2-21.

Internal Component Variables

You can also declare an internal component variable as a value with unit on page 2-5. You can use such internal variables throughout the component file, for example, in the `equations` section or in the intermediate term declarations. Component variables are also used in the model initialization process, as described in "Variable Priority for Model Initialization" on page 2-7.

The following example declares and initializes three variables:

```
variables
    f = { 0, 'N' };   % Force
    v = { 0, 'm/s' }; % Velocity
    x = { 0, 'm' };   % Spring deformation
end
```

Force and velocity are the component Through and Across variables, later to be connected to the domain Through and Across variables using the `branches` section. Spring deformation is an internal component variable, to be used for model initialization.

You can declare internal component variables of type integer or real as event variables by setting the `Event=true` attribute. For more information, see "Event Variables" on page 2-52.

Variable Priority for Model Initialization

When you generate a custom Simscape block from a component file, the **Variables** tab of this block will list all the public variables specified in the underlying component file, along with the initialization priority, target initial value, and unit of each variable. The block user can change the variable priority and target, prior to simulation, to affect the model initialization. For more information, see “Variable Initialization”.

The default values for variable priority, target value, and unit come from the variable declaration in the component file. Specifying an optional comment lets you control the variable name in the block dialog box. For more information, see “Specify Meaningful Names for the Block Parameters and Variables” on page 4-36.

Note For variables with temperature units, there is an additional consideration of whether to apply linear or affine conversion when the block user changes the unit in the **Variables** tab of the block dialog box. Use the `Conversion` attribute in the same way as for the block parameters. For details, see “Parameter Units” on page 2-11.

In most cases, it is sufficient to declare a variable just as a value with unit on page 2-5, omitting its priority, which is equivalent to `priority = priority.none`. The block user can set the variable priority, as needed, in the **Variables** tab of the block dialog box prior to simulation.

In some cases, however, setting a variable to a certain priority by default is essential to the correct operation of the component. To specify a high or low default priority for a component variable, declare the variable as a field array. For example, the following declaration initializes variable `x` (spring deformation) as 0 mm, with high priority:

```
variables
    x = { value = { 0 , 'm' }, priority = priority.high }; % Spring deformation
end
```

In this case, the **Spring deformation** variable will appear in the **Variables** tab of the block dialog box with the default priority `High` and the default target value and unit `0 mm`, but the block user can change the variable priority and target as usual.

If you want a variable to always have high initialization priority, without letting the block user to change it, declare the variable as private:

```
variables(Access=private)
  x = { value = { 0 , 'm' }, priority = priority.high };
end
```

In this case, the block user does not have control over the variable priority or initialization target, because private variables do not appear in the **Variables** tab of the block dialog box.

If you want the variable to always have a certain initialization priority, such as `High`, but let the block user specify the target value, declare the variable as private and tie it to an initialization parameter:

```
parameters
  p = { 0, 'm' }; % Initial deformation
end
variables(Access=private)
  x = {value = p, priority = priority.high };
end
```

In this case, the value of the **Initial deformation** parameter, specified by the block user, is assigned as the initial target to variable `x`, with high initialization priority. Depending on the results of the solve, this target may or may not be satisfied when the solver computes the initial conditions for simulation. For more information, see “Initial Conditions Computation”.

For composite components, member components are declared as hidden and therefore their variables do not appear in the **Variables** tab of the block dialog box. However, you can use a top-level parameter to let the block user specify the initial target value of a member component variable. For more information, see “Specifying Initial Target Values for Member Variables” on page 2-62.

Nominal Value and Unit for a Variable

Nominal values provide a way to specify the expected magnitude of a variable in a model, similar to specifying a transformer rating, or setting a range on a voltmeter. For more information, see “System Scaling by Nominal Values”.

Each model has an underlying table of nominal value-unit pairs. In general, all variables in a model are scaled based on the nominal value corresponding to their physical unit. You can override this scaling for an individual variable in a component file by providing a nominal value and unit as a variable declaration attribute.

```
variables
    x = { value = { value , 'unit' }, nominal = {value, 'unit'} };
end
```

When you generate a custom Simscape block from a component file, nominal value and unit form the `nominal` declaration attribute translate into default values for block parameters *`x`*`_nominal` and *`x`*`_nominal_unit` (where *`x`* is the variable name).

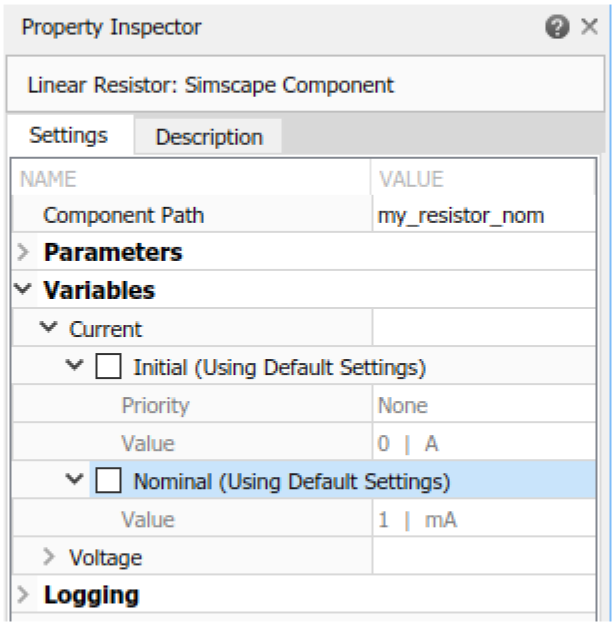
For example, this variable declaration:

```
variables
    i = { value = { 0 , 'A' }, nominal = {1, 'mA'} }; % Current
end
```

produces the following default values for block parameters:

- `i_nominal_value`, with a value of `'1'`
- `i_nominal_unit`, with a value of `'mA'`

and looks like this in the Property Inspector.



Property Inspector

Linear Resistor: Simscape Component

Settings | Description

NAME	VALUE
Component Path	my_resistor_nom
Parameters	
Variables	
Current	
Initial (Using Default Settings)	
Priority	None
Value	0 | A
Nominal (Using Default Settings)	
Value	1 | mA
Voltage	
Logging	

Note It is recommended that you use the `nominal` attribute sparingly. The default nominal values, which come from the model value-unit table, are suitable in most cases. The block user can also modify the nominal values and units for individual blocks by using either the Property Inspector or `set_param` and `get_param` functions, if needed. For more information, see “Modify Nominal Values for a Block Variable”.

See Also

Related Examples

- “Declare a Spring Component” on page 2-19
- “Declare Through and Across Variables for a Domain” on page 2-6
- “Declare Component Parameters” on page 2-11
- “Declaring Domain Parameters” on page 2-98
- “Declare Component Nodes” on page 2-14
- “Declare Component Inputs and Outputs” on page 2-16

More About

- “Declaring Domains and Components” on page 2-3

Declare Component Parameters

In this section...

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 on page 2-5. 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 and Variables" on page 4-36.

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
```

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 equations and other sections of a component file, 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"). 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 2-104).

Note Member attributes apply to a whole declaration block. 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
```

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

Public component variables also appear in the block dialog box, on the **Variables** tab, because they are used for model initialization. These variables therefore compete with each other and with the block parameter names in the Simulink namespace. If a component has a public variable and a parameter whose names differ only by case, such as

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

you will not be able to generate a block from this component. As a possible workaround, you can declare the variable as private or hidden. In this case, the variable does not appear on the **Variables** tab of the resulting block dialog, and therefore there is no namespace conflict. However, if you want to be able to use the variable in the model initialization process, keep it public and change its name, or the name of the parameter.

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

See Also

Related Examples

- “Declare a Spring Component” on page 2-19
- “Declare Component Variables” on page 2-7
- “Declare Component Nodes” on page 2-14
- “Declare Component Inputs and Outputs” on page 2-16

More About

- “Declaring Domains and Components” on page 2-3
- “Enumerations” on page 3-15

Declare 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”.

When declaring nodes in a component, you have to associate them with an existing domain. Once a node is associated with a domain, it:

- Carries each of the domain Across variables as a measurable quantity
- Writes a conserving equation for each of the domain Through variables

For more information, see “Define Relationship Between Component Variables and Nodes” on page 2-21.

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 “Building Custom Block Libraries” on page 4-25.

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 4-38. 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
```

See Also

Related Examples

- “Declare a Spring Component” on page 2-19
- “Declare a Mechanical Rotational Domain” on page 2-18
- “Declare Through and Across Variables for a Domain” on page 2-6
- “Declare Component Variables” on page 2-7
- “Declare Component Parameters” on page 2-11
- “Declare Component Inputs and Outputs” on page 2-16

More About

- “Declaring Domains and Components” on page 2-3

Declare 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 can be defined as:

- A value with unit on page 2-5, where `value` can be a scalar, vector, or matrix. For a vector or a matrix, all signals have the same unit.
- An untyped identifier, to facilitate unit propagation.

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 4-38.

This 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
```

You can also reference component parameters in input and output declarations. For example, you can control the signal size by using a block parameter:

```
component MyTransformer
     parameters
         N = 3; % Number of windings
     end
     inputs
         I = {zeros(N, 1), 'A'};
     end
     ....
 end
```

The following example declares an input port `I` and output port `O` as untyped identifiers. In the block diagram, the output port will be located on the right side of the block icon. The block propagates the unit and size of the physical signal. For more information, see “Physical Signal Unit Propagation”.

```
 inputs
   I;
 end
 outputs
   O; % :right
 end
```

See Also

Related Examples

- “Declare a Spring Component” on page 2-19
- “Declare Component Variables” on page 2-7
- “Declare Component Parameters” on page 2-11
- “Declare Component Nodes” on page 2-14

More About

- “Declaring Domains and Components” on page 2-3
- “Physical Signal Unit Propagation”

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 6-2.

In a component, each node associated with this domain will:

- Carry a measurable variable `w` (angular velocity)
- Conserve variable `t` (torque)

For more information, see “Define Relationship Between Component Variables and Nodes” on page 2-21.

See Also

Related Examples

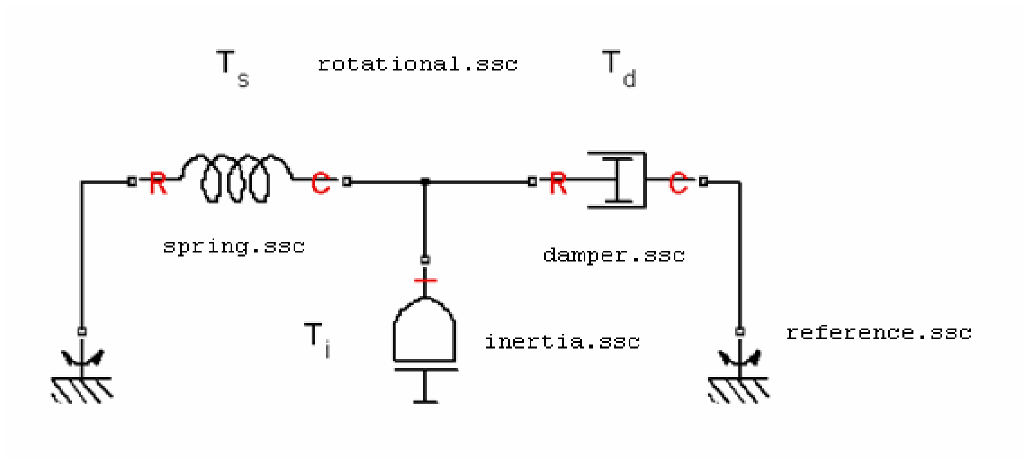
- “Declare Through and Across Variables for a Domain” on page 2-6
- “Declaring Domain Parameters” on page 2-98

More About

- “Declaring Domains and Components” on page 2-3

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 2-18). 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
  % branches 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, see “Mechanical Component — Spring” on page 2-90.

See Also

Related Examples

- “Declare a Spring Component” on page 2-19
- “Declare Component Variables” on page 2-7
- “Declare Component Parameters” on page 2-11
- “Declare Component Nodes” on page 2-14
- “Declare Component Inputs and Outputs” on page 2-16

More About

- “Declaring Domains and Components” on page 2-3

Define Relationship Between Component Variables and Nodes

In this section...

Connecting Component Variables to the Domain

After you declare the component Through and Across variables on page 2-7, you need to connect them to the domain Through and Across variables. You do this by establishing the relationship between the component variables and its nodes, which carry the Through and Across variables for the domain:

- To establish the relationship between the Through variables, use the `branches` section of the component file. If the component has multiple nodes, indicate branches by writing multiple statements in the `branches` section. For syntax and examples, see the `branches on page 5-9` reference page.
- To establish the relationship between the Across variables, use the `equations` section of the component file. Add an equation that connects the component Across variable with the respective variables at the component nodes. If there is more than one Across variable, add multiple equations, connecting each variable with its respective nodes. The `equations` section can also contain other equations that define the component action. For more information, see “Defining Component Equations” on page 2-24.

Workflow from Domain to Component

Propagate the domain Through and Across variables into a component.

1. Declare the Across and Through variables in a domain file (or use an existing domain; for a complete listing of the Foundation domains, see “Foundation Domain Types and Directory Structure” on page 6-2).

   For example, the following domain file, named `rotational.ssc`, declares angular velocity, `w`, as an Across variable and torque, `t`, 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
```

2 Declare the nodes in a component file and associate them with the domain, for example:

```
nodes
    node1 = MyPackage.rotational;
    node2 = MyPackage.rotational;
end
```

Once a node is associated with a domain, it:

- Carries each of the domain Across variables as a measurable quantity. In this example, each of the nodes carries one Across variable, `w`.
- Writes a conserving equation for each of the domain Through variables. In this example, there is one Through variable, `t`, and therefore each node writes one conserving equation. A conserving equation is a sum of terms that is set to zero `(node.t == 0)`. The `branches on page 5-9` section in the component file establishes the terms that are summed to zero at the node.

3 Declare the corresponding variables in the component file, for example:

```
variables
    w = { 1 , 'rad/s' };   % angular velocity
    t = { 1 , 'N*m' };     % torque
end
```

The names of the component variables do not have to match those of the domain Across and Through variables, but the units must be commensurate. At this point, there is no connection between the component variables and the domain variables.

4 Establish the relationship between the Through variables by using the `branches` section of the component file. For example:

```
branches
    t : node1.t -> node2.t;    % t - Through variable from node1 to node2
end
```

This branch statement declares that `t` flows from `node1` to `node2`. Therefore, `t` is subtracted from the conserving equation identified by `node1.t`, and `t` is added to the conserving equation identified by `node2.t`. For more information and examples, see the `branches on page 5-9` reference page.

5 Establish relationship between the Across variables in the `equations` section of the component file, for example, by adding the following equation:

```
equations
     w == node1.w - node2.w;       % w - Across variable between node1 and node2
     [...]      % more equations describing the component behavior, as necessary
end
```

Connecting One Through and One Across Variable

In this example, `r` and `c` are rotational nodes, while `t` and `w` are component variables for torque and angular velocity, respectively. The relationship between the variables and nodes is established in the `branches` and the `equations` sections:

```
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
  branches
    t : r.t -> c.t;       % t - Through variable from r to c
  end
  equations
    w == r.w - c.w;       % w - Across variable between r and c
    [...]                 % more equations here
  end
end
```

Connecting Two Through and Two Across Variables

This example shows setting up the Across and Through variables of a component with two electrical windings, such as a transformer or mutual inductor. The component has four electrical nodes, and each winding has its own voltage and current variables. The relationship between the variables and nodes is established in the `branches` and the `equations` sections:

```
component two_windings
  nodes
    p1 = foundation.electrical.electrical;
    n1 = foundation.electrical.electrical;
    p2 = foundation.electrical.electrical;
    n2 = foundation.electrical.electrical;
  end
  [...]
  variables
    i1 = { 0, 'A' };
    v1 = { 0, 'V' };
    i2 = { 0, 'A' };
    v2 = { 0, 'V' };
  end
  [...]
  branches
    i1 : p1.i -> n1.i;   % Current through first winding
    i2 : p2.i -> n2.i;   % Current through second winding
  end
  equations
    v1 == p1.v - n1.v;   % Voltage across first winding
    v2 == p2.v - n2.v;   % Voltage across second winding
    [...]                % more equations here
  end
end
```

Defining Component Equations

In this section...

Equation Section Purpose

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. The equation section of a Simscape file is executed throughout the simulation.

Note You can also specify equations that are executed during model initialization only, by using the `(Initial=true)` attribute. For more information, see “Initial Equations” on page 2-31.

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 2-24). 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.

For a list of MATLAB functions that you can use in the equation section, see Supported Functions.

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 on page 2-25 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 on page 2-24 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).

Working with Physical Units in Equations

In Simscape language, you declare members (such as parameters, variables, inputs, and outputs) as value with unit on page 2-5, 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.

See Also

Related Examples

- “Simple Algebraic System” on page 2-29
- “Use Simulation Time in Equations” on page 2-30

More About

- “Using Conditional Expressions in Equations” on page 2-33
- “Using Intermediate Terms in Equations” on page 2-35
- “Using Lookup Tables in Equations” on page 2-46
- “Programming Run-Time Errors and Warnings” on page 2-48

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
  outputs
    x = 0;
    y = 0;
  end
  equations
    y == x^2;        % y = x^2
    x == 2 * y - 1;  % x = 2 * y - 1
  end
end
```

See Also

Related Examples

- “Use Simulation Time in Equations” on page 2-30

More About

- “Defining Component Equations” on page 2-24
- “Using Conditional Expressions in Equations” on page 2-33
- “Using Intermediate Terms in Equations” on page 2-35
- “Using Lookup Tables in Equations” on page 2-46
- “Programming Run-Time Errors and Warnings” on page 2-48

Use 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
```

See Also

Related Examples

- “Simple Algebraic System” on page 2-29

More About

- “Defining Component Equations” on page 2-24
- “Using Conditional Expressions in Equations” on page 2-33
- “Using Intermediate Terms in Equations” on page 2-35
- “Using Lookup Tables in Equations” on page 2-46
- “Programming Run-Time Errors and Warnings” on page 2-48

Initial Equations

Regular equations are executed throughout the simulation. The `(Initial=true)` attribute lets you specify additional equations that are executed during model initialization only.

Regular component equations alone are not sufficient to initialize a DAE system. Consider a system with *n* continuous differential variables and *m* continuous algebraic variables. For simulation, this system has *n*+*m* degrees of freedom and must provide *n*+*m* equations. The initialization problem has up to *n* additional unknowns that correspond to the derivative variables. These additional unknowns can be satisfied when you specify initial targets for block variables. Initial equations provide another way to initialize a system.

In general, the maximum number of high-priority targets you can specify is equal to the number of additional unknowns in the initialization problem. Besides the unknowns from differential variables, the initialization problem also has one more unknown for each event variable. These additional unknowns determine the maximum combined number of initial equations and high-priority variable targets. If there are too many high-priority targets, these cannot all be met. For more information, see “Block-Level Variable Initialization”.

Because the default value of the `Initial` attribute for equations is `false`, you can omit this attribute when declaring regular equations:

```
equations (Initial = true)  % initial equations
  [...]
end

equations (Initial = false) % regular equations
  [...]
end

equations % regular equations
  [...]
end
```

The syntax of initial equations is the same as that of regular equations, except:

- `der(`*`x`*`)` in initial equations is treated as an unknown value and is solved for during initialization.
- `delay` and `integ` operators are disallowed.

When you include `assert` constructs in initial equations, their predicate conditions are checked only once, after solving for initial conditions (before the start of simulation, see “Initial Conditions Computation”). Use these assertions to safeguard against the model initializing with nonphysical values. For more information, see “Programming Run-Time Errors and Warnings” on page 2-48.

A common use case for specifying initial equations is to initialize a system in steady state, for example:

```
component C

    parameters
        a = {-5, '1/s'};
        b = {-2, '1/s'};
    end

    outputs
```

```
            x = 5;
            y = 10;
        end

        equations
           der(x) == a*x + b*y;
           der(y) == b*y;
        end

        equations(Initial=true)
           der(x) == 0;
           der(y) == 0;
        end

    end
```

At initialization time, the equations are:

```
der(x) == 0;
der(y) == 0;
der(x) == a*x + b*y;
der(y) == b*y;
```

For the rest of the simulation, the equations are:

```
der(x) == a*x + b*y;
der(y) == b*y;
```

Note When you initialize a model from an operating point, especially one that was generated from logged simulation data, the operating point is likely to contain all the necessary high-priority targets and applying initial equations would result in an over-specified model. Therefore, if you initialize a model from an operating point, the solver ignores all the initial equations that contain variables present in the operating point data. Initial equations for other variables are not affected: for example, if you add a block to the model after extracting the operating point data, initial equations for this block will be executed at initialization time. For more information, see “Using Operating Point Data for Model Initialization”.

See Also

More About

- “Defining Component Equations” on page 2-24
- “Using Conditional Expressions in Equations” on page 2-33
- “Using Intermediate Terms in Equations” on page 2-35
- “Using Lookup Tables in Equations” on page 2-46
- “Programming Run-Time Errors and Warnings” on page 2-48

Using Conditional Expressions in Equations

In this section...

Statement Syntax

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.

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
```

See Also

More About

- “Defining Component Equations” on page 2-24
- “Using Intermediate Terms in Equations” on page 2-35
- “Using Lookup Tables in Equations” on page 2-46
- “Programming Run-Time Errors and Warnings” on page 2-48

Using Intermediate Terms in Equations

In this section...

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, there are two ways that you can define intermediate terms for use in equations:

- `intermediates` section — Declare reusable named intermediate terms in the `intermediates` section in a component or domain file. You can reuse these intermediate terms in any equations section within the same component file, in an enclosing composite component file, or in any component that has nodes of that domain type.
- `let` expressions in the `equations` section — Declare intermediate terms in the declaration clause and use them in the expression clause of the same `let` expression. Use this method if you need to define intermediate terms of limited scope, for use in a single group of equations. This way, the declarations and equations are close together, which improves code readability.

Another advantage of using named intermediate terms instead of `let` expressions is that you can include named intermediate terms in simulation data logs.

The following example shows the same Darcy-Weisbach equation with intermediate terms 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
  intermediates
    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 diameter
    V    = q / A;                % Flow velocity
  end
  equations
      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));
```

When you use this component in a model and log simulation data, the logs will include data for the four intermediate terms, with their descriptive names (such as `Darcy friction factor`) shown in the Simscape Results Explorer.

Declaring and Using Named Intermediate Terms

The `intermediates` section in a component file lets you define named intermediate terms for use in equations. Think of named intermediate terms as of defining an alias for an expression. You can reuse it in any equations section within the same file or an enclosing composite component. When an intermediate term is used in an equation, it is ultimately substituted with the expression that it refers to.

You can also include an `intermediates` section in a domain file and reuse these intermediate terms in any component that has nodes of that domain type.

Syntax Rules and Restrictions

You declare an intermediate term by assigning a unique identifier on the left-hand side of the equal sign (=) to an expression on the right-hand side of the equal sign.

The expression on the right-hand side of the equal sign:

- Can refer to other intermediate terms. For example, in the Darcy-Weisbach equation, the identifier `Re_d` (Reynolds number) is used in the expression declaring the identifier `f` (Darcy friction factor). The only requirement is that these references are acyclic.

- Can refer to parameters, variables, inputs, outputs, member components and their parameters, variables, inputs, and outputs, as well as Across variables of domains used by the component nodes.
- Cannot refer to Through variables of domains used by the component nodes.

You can use intermediate terms in equations, as described in “Use in Equations” on page 2-37. However, you cannot access intermediate terms in the `setup` function.

Intermediate terms can appear in simulation data logs and Simscape Results Explorer, as described in “Data Logging” on page 2-38. However, intermediate terms do not appear in:

- Variable Viewer
- Statistics Viewer
- Operating Point data
- Block dialog boxes and Property Inspector

Use in Equations

After declaring an intermediate term, you can refer to it by its identifier anywhere in the equations section of the same component. For example:

```
component A
  [...]
  parameters
    p1 = { 1, 'm' };
  end
  variables
    v1 = { 0, 'm' };
    v2 = { 0, 'm^2' };
  end
  intermediates
    int_expr = v1^2 * pi / p1;
  end
  equations
      v2 == v1^2 + int_expr;
  end
end
```

You can refer to a public intermediate term declared in a member component in the equations of an enclosing composite component. For example:

```
component B
  [...]
  components
    comp1 = MyPackage.A;
  end
  variables
    v1 = { 0, 'm^2' };
  end
  [...]
  equations
      v1 == comp1.int_expr;
  end
end
```

Similarly, you can refer to an intermediate term declared in a domain in the equations section of any component that has nodes of this domain type. For example:

```
domain D
  [...]
  intermediates
    int_expr = v1 / sqrt(2);
  end
  [...]
end

component C
  [...]
  nodes
    n = D;
  end
  variables
    v1 = { 0, 'V' };
  end
  [...]
  equations
      v1 == n.int_expr;
  end
end
```

Accessibility of intermediate terms outside of the file where they are declared is governed by their `Access` attribute value. For mode information, see "Attribute Lists" on page 2-103.

Data Logging

Intermediate terms with `ExternalAccess` attribute values of `modify` or `observe` are included in simulation data logs. For mode information, see "Attribute Lists" on page 2-103.

If you specify a descriptive name for an intermediate term, this name appears in the status panel of the Simscape Results Explorer.

For example, you declare the intermediate term `D_h` (hydraulic diameter) as a function of the orifice area:

```
component E
  [...]
  intermediates
    D_h  = sqrt( 4.0 * A / pi ); % Hydraulic diameter
  end
  [...]
end
```

When you use a block based on this component in a model and log simulation data, selecting `D_h` in the Simscape Results Explorer tree on the left displays a plot of the values of the hydraulic diameter over time in the right pane and the name `Hydraulic diameter` in the status panel at the bottom. For more information, see "About the Simscape Results Explorer".

Using the let Expressions

`let` expressions provide another way to define intermediate terms for use in one or more equations. Use this method if you need to define intermediate terms of limited scope, for use in a single group of equations. This way, the declarations and equations are close together, which improves file readability.

The following example shows the same Darcy-Weisbach equation as in the beginning of this topic but with intermediate terms written out using the `let` expression:

```
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 diameter
      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));
```

However, in this case the four intermediate terms do not appear in logged simulation data.

Syntax Rules of let Expressions

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 example with the Darcy-Weisbach equation, 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 than a single identifier, in the declarative clause of a `let` expression:

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

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

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

then the second semantic translation becomes

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

and the final substitution on this example yields:

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

See Also

`intermediates`

More About

- “Defining Component Equations” on page 2-24
- “Using Conditional Expressions in Equations” on page 2-33
- “Using Lookup Tables in Equations” on page 2-46
- “Programming Run-Time Errors and Warnings” on page 2-48

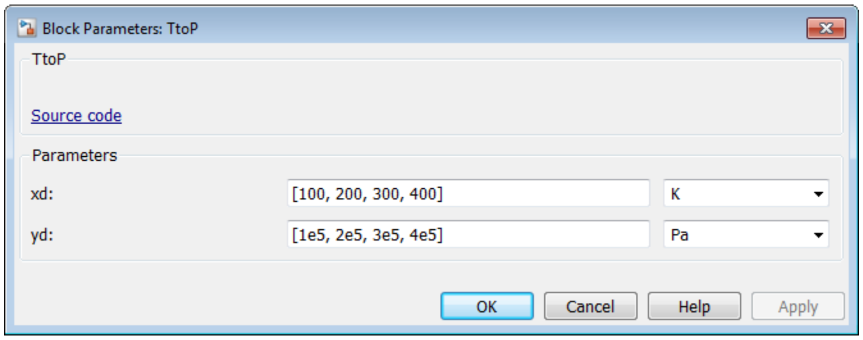
Using Lookup Tables in Equations

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, two-dimensional, or three-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.



xd: [100, 200, 300, 400] K
yd: [1e5, 2e5, 3e5, 4e5] Pa

Note Currently, you cannot use variable-size parameters in the `equations` section outside of the `tablelookup` function.

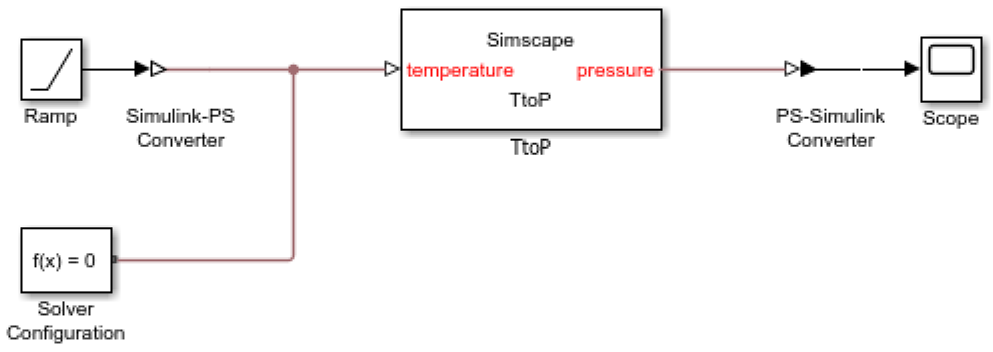
To avoid repeating the same variable-size parameter declarations in each component that needs to use them in its `tablelookup` function, you can declare variable-size domain parameters and propagate them to components for interpolation purposes. For more information, see “Propagation of Domain Parameters” on page 2-98.

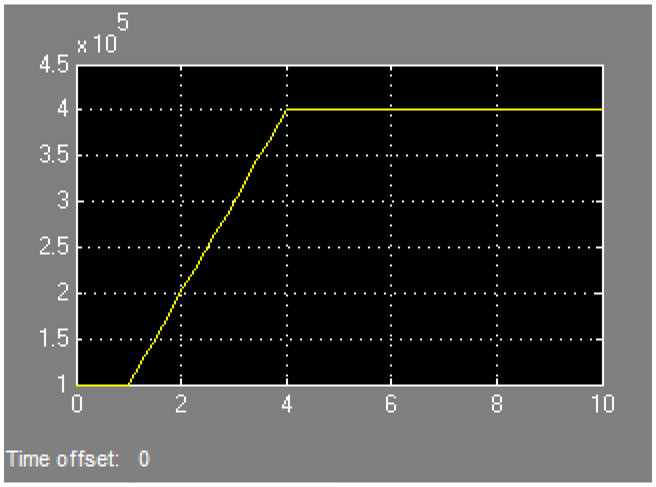
The following rules apply to the one-dimensional arrays `xd` and `yd`:

- The two arrays must be of the same size.

- For smooth 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.

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.

See Also

More About

- "Defining Component Equations" on page 2-24
- "Using Conditional Expressions in Equations" on page 2-33
- "Using Intermediate Terms in Equations" on page 2-35
- "Programming Run-Time Errors and Warnings" on page 2-48

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 optional `Action` attribute of the `assert` construct specifies whether simulation stops when the predicate condition is violated, continues with a warning, or ignores the violation.

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

  branches
    i : p.i -> n.i;
  end

  equations
    assert( R >= 0, 'Negative resistance is not modeled' );
    v == p.v - n.v;
    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.

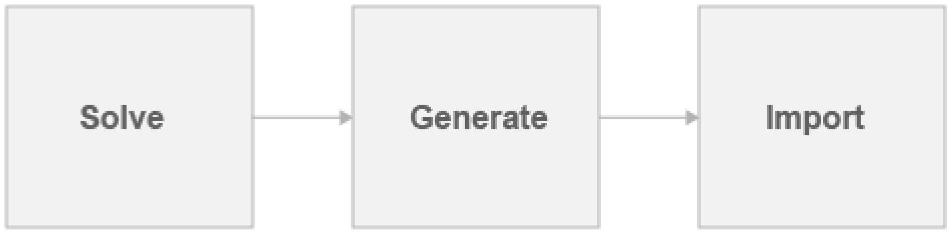
See Also

More About

- "Defining Component Equations" on page 2-24
- "Using Conditional Expressions in Equations" on page 2-33
- "Using Intermediate Terms in Equations" on page 2-35
- "Using Lookup Tables in Equations" on page 2-46

Import Symbolic Math Toolbox Equations

When designing a Simscape language component, you can use Symbolic Math Toolbox software to solve the physical equations and generate code in the format appropriate for the Simscape language equation section. Then, import the results by copying and pasting them into the equation section of a component file and declaring all the symbolic variables used in these equations.



Suppose, you want to generate a Simscape equation from the solution of the following ordinary differential equation. As a first step, use the `dsolve` function to solve the equation:

```
syms a y(t)
Dy = diff(y);
s = dsolve(diff(y, 2) == -a^2*y, y(0) == 1, Dy(pi/a) == 0);
s = simplify(s)
```

The solution is:

```
s =
cos(a*t)
```

Then, use the `simscapeEquation` function to rewrite the solution in the Simscape language equation format:

```
simscapeEquation(s)
```

`simscapeEquation` generates the following code:

```
ans =
s == cos(a*time);
```

Copy and paste the generated code into the equation section of a component file:

```
component MyComponent

  equations
        s == cos(a*time);
  end
end
```

Make sure the declaration section of the component file contains all the symbolic variables used in these equations. You can declare these symbolic variables as Simscape language variables, parameters, inputs, or outputs, depending on their physical function and your intended block design.

```
component MyComponent
  inputs
    a = {1,'m/s'};
  end
  outputs
    s = {0,'m'};
  end
  equations
        s == cos(a*time);
  end
end
```

See Also

Related Examples

- "Use Simulation Time in Equations" on page 2-30

More About

- "Get Started with Symbolic Math Toolbox" (Symbolic Math Toolbox)
- "Generate Simscape Equations from Symbolic Expressions" (Symbolic Math Toolbox)

Discrete Event Modeling

In this section...

Physical modeling, in general, involves continuous variables and equations. In some cases, however, you can simplify the mathematical model of the system and improve simulation performance by treating certain changes in system behavior as discrete. Such an idealization assumes that system variables may only change values instantaneously and discontinuously at specific points in time.

An event is a conceptual notation that denotes a change of state in a system. Event modeling lets you perform discrete changes on continuous variables. The two most common applications of event modeling are:

- Trigger-and-hold mechanism, such as a triggered delay. For example, a component has two inputs: `u` and `x` (triggered signal), and one output `y`. When and only when the triggered signal `x` changes value from false to true, output `y` is reset to the value of `u` at current time. `y` remains unchanged all other times.
- Enabled component, acting on a principle similar to Simulink enabled subsystem. That is, the component has a control signal as input. If the control signal has a positive value, then the component holds certain states to the most recent value, or resets them. When the control signal is negative, the states change according to component equations.

The following constructs in Simscape language let you perform event modeling: event variables, `events` section, `when` clause, and `edge` operator.

Event Variables

Event variables are piecewise constant, that is, they change values only at event instants, and keep their values constant between events. You can declare internal component variables of type integer or real as event variables by setting the `Event=true` attribute.

For example, the following code declares two event variables: `x` (type real) and `d` (type integer).

```
variables (Event=true)
   x = 0;
   d = int32(0);
end
```

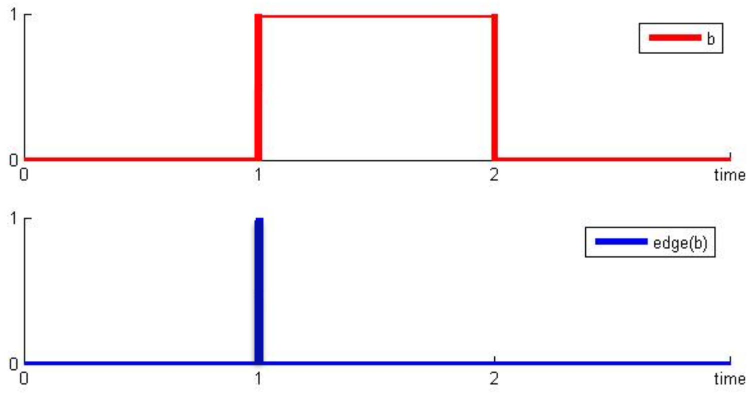
You can initialize event variables by using the `initialevent` operator. You can also initialize event variables the same way as continuous variables, by setting their target values and priorities in the member declaration block. For more information, see `initialevent`.

You update the values of the event variables in the `events` section of the component file, by using the `when` clause.

Event Data Type and edge Operator

The `edge` operator takes a scalar Boolean expression as input. It returns true, and triggers an event, when and only when the input argument changes value from false to true. The return type of `edge` is event type. Event data type is a special category of Boolean type, which returns true only instantaneously, and returns false otherwise.

The following graphic illustrates the difference between Boolean and event data types.



`edge(b)` returns true only when `b` changes from false to true.

To trigger an event on the falling edge of condition `b`, use `edge(~b)`.

The data derivation rules between Boolean and event data types are:

- `edge`(`boolean`) is `event`
- ~`event` is `boolean`
- (`event` && `event`) is `event`
- (`event` && `boolean`) is `event`
- (`event` || `event`) is `event`
- (`event` || `boolean`) is `boolean`

You use the `edge` operator to define event predicates in `when` clauses.

Events Section and when Clause

The `events` section in a component file manages the event updates. The `events` section can contain only `when` clauses. The order of `when` clauses does not matter.

The `when` clause serves to update the values of the event variables. The syntax is

```
when EventPredicate
  var1 = expr1;
  var2 = expr2;
  ...
end
```

`EventPredicate` is an expression that defines when an event occurs. It must be an expression of event data type, as described in “Event Data Type and edge Operator” on page 2-52.

The variables in the body of the `when` clause must be declared as event variables. When the event predicate returns true, all the variables in the body of the `when` clause simultaneously get updated to the new values.

The order of the variable assignments in the body of the `when` clause does not matter, because all updates happen simultaneously. For example, if `d1` and `d2` are event variables initialized to 0,

```
when edge(time>1.0)
   d1 = d2 + 1;
   d2 = d1 + 1;
 end
```

is equivalent to:

```
when edge(time>1.0)
   d2 = d1 + 1;
   d1 = d2 + 1;
end
```

After the event, both `d1` and `d2` have a new value of 1, because they were both simultaneously updated by adding 1 to the old value of 0.

A `when` clause cannot update an event variable more than once within the same assignments list. However, two independent `when` clauses also may not update the same event variable. You must use an `elsewhen` branch to do this.

Branching of the elsewhen Clauses

A `when` clause can optionally have one or more `elsewhen` branches:

```
when EventPredicate
  var1 = expr1;
  var2 = expr2;
  ...
elsewhen EventPredicate
  var1 = expr3;
  ...
end
```

Note The default `else` branch in a `when` clause is illegal.

A common usage of `elsewhen` branches is to prioritize events. If multiple predicates become true at the same point in time, only the branch with the highest precedence is activated. The precedence of the branches in a `when` clause is determined by their declaration order. That is, the `when` branch has the highest priority, the last `elsewhen` branch has the lowest priority.

See Also

Related Examples

- “Triggered Delay Component” on page 2-55
- “Enabled Component” on page 2-56

Triggered Delay Component

The following example implements a triggered delay component:

```
component Triggered
   inputs
      u = 0; % input signal
      triggered = 0; % control signal
   end
   variables(Event=true)
      x = 0;
   end
   outputs
      y = 0;
   end
   events
      when edge(triggered>0)
        x = u;
      end
   end
   equations
      y == x;
   end
end
```

When the control signal becomes positive, the event variable `x` gets updated to the current value of the input signal `u`. Output `y` outputs the value of `x`. Therefore, the output signal `y` gets updated to the current value of the input signal `u` on the rising edge of the control signal, and then holds that value between the events.

See Also

Related Examples

- “Enabled Component” on page 2-56

More About

- “Discrete Event Modeling” on page 2-52

Enabled Component

The following example implements a component similar to a Simulink enabled subsystem:

```
component EnabledComponent
    inputs
        enabled = 0;     % control signal
        u = 0;         % input signal
    end
    variables (Event=true)
        x = 0;         % state to hold output if necessary
    end
    outputs
        y = 0;         % output
    end
    parameters
        held = true;     % set true for held or false for reset
        y_init = 0;
    end
    events
        when edge(held && ~(enabled>0))
            x = u;    % if necessary, hold input on falling edge
        end
    end
    equations
        if enabled > 0
            y == u;
        elseif held==true
            y == x;
        else        % not enabled and not held
            y == y_init;
        end
    end
end
```

The component has two inputs: control signal `enabled` and data signal `u`.

The block operation depends on the value of the `held` parameter: if it is `true`, then the event variable `x` assumes the value of the input data signal `u` on the falling edge of the control signal.

As long as the control signal has a positive value, the output `y` matches the input data signal `u`. When the control signal is negative:

- If `held` is `true`, the output port `y` outputs the most recent held value of the event variable.
- If `held` is `false`, the output resets to the initial value, specified by the `y_init` parameter.

See Also

Related Examples

- "Triggered Delay Component" on page 2-55

More About

- “Discrete Event Modeling” on page 2-52

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.

See Also

Related Examples

- “Composite Component — DC Motor” on page 2-95
- “Composite Component Using import Statements” on page 2-110
- “Component Variants — Series RLC Branch” on page 2-83
- “Segmented Pipeline Using Component Array” on page 3-32
- “Case Study — Battery Pack with Fault Using Arrays” on page 3-34

More About

- “Declaring Member Components” on page 2-59
- “Parameterizing Composite Components” on page 2-60
- “Specifying Initial Target Values for Member Variables” on page 2-62
- “Specifying Component Connections” on page 2-64
- “Importing Domain and Component Classes” on page 2-108
- “Defining Component Variants” on page 2-75
- “Component Arrays” on page 3-29
- “Converting Subsystems into Composite Components” on page 2-70

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 `ExternalAccess` attribute value set to `observe` (for more information on member attributes, see "Attribute Lists" on page 2-103).

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 "Building Custom Block Libraries" on page 4-25.

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

```
components(ExternalAccess=observe)
    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(ExternalAccess=observe)
    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.

See Also

Related Examples

- "Composite Component — DC Motor" on page 2-95

More About

- "Parameterizing Composite Components" on page 2-60
- "Specifying Initial Target Values for Member Variables" on page 2-62
- "Specifying Component Connections" on page 2-64
- "Importing Domain and Component Classes" on page 2-108

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 “Declare Component Parameters” on page 2-11.

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(ExternalAccess=observe)
     r1 = foundation.electrical.elements.resistor(R = p1);
     [...]
  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.

Caution on Using setup to Parameterize Composite Components

You can establish the connection of a top-level parameter with a member component parameter either in the `components` declaration block, or later, in the `setup` section. Starting in R2019a, using `setup` is not recommended. If you have legacy code using the `setup` function, update it to use parameter assignment in the `components` block instead. For example, this code is equivalent to the example above:

```
component MyCompositeModel
[...]
  parameters
     p1 = {10, 'Ohm'};
     [...]
  end
  components(ExternalAccess=observe)
     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(ExternalAccess=observe)
    C = {1 , 'F'};
  end
  components(ExternalAccess=observe)
    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
```

See Also

Related Examples

- “Composite Component — DC Motor” on page 2-95

More About

- “Declaring Member Components” on page 2-59
- “Specifying Initial Target Values for Member Variables” on page 2-62
- “Specifying Component Connections” on page 2-64

Specifying Initial Target Values for Member Variables

Member components have to be declared with `ExternalAccess=observe`, and therefore their variables do not appear in the **Variables** tab of the top-level block dialog box. However, if a certain member component variable is important for initialization, you can tie its value to an initialization parameter in the top-level `parameters` declaration block. In this case, the block user will be able to adjust the initial target value of the member component variable from the top-level block dialog box when building and simulating a model.

Note The block user cannot change the initialization priority of the member component variable. You specify the variable initialization priority when you declare the member component. The syntax is the same as described in “Variable Priority for Model Initialization” on page 2-7.

For example, you have a composite DC Motor block (similar to the one described in “Composite Component — DC Motor” on page 2-95) and want the block user to specify the initial target value for the inductor current, with low priority. The following code includes a Foundation library Inductor block in your custom component file, with the ability to control its inductance at the top level (by using the **Rotor Inductance** block parameter) and also to specify a low-priority initial target for the inductor current variable:

```
component DCMotor2
[...]
  parameters
     rotor_inductance = { 12e-6, 'H' };    % Rotor Inductance
     i0 = { 0, 'A' };  % Initial current target for Rotor Inductor
     [...]
  end
  components(ExternalAccess=observe)
     rotorInductor = foundation.electrical.elements.inductor(l = rotor_inductance,
                                  i_L = {value = i0, priority = priority.low});
     [...]
  end
[...]
end
```

In this case, the block user can specify a value for the **Initial current target for Rotor Inductor** parameter, which appears in the block dialog box of the composite component. This value gets assigned as the initial target to variable `i_L` (**Initial current** variable of the member Inductor block), with low initialization priority. Depending on the results of the solve, the target may or may not be satisfied when the solver computes the initial conditions for simulation. For more information, see “Block-Level Variable Initialization”.

You can use an alternative syntax that lets you assign the variable value and priority data fields separately, using the dot notation. For example, the following statement:

```
      rotorInductor = foundation.electrical.elements.inductor(l = rotor_inductance,
                                  i_L.value = i0, i_L.priority = priority.low);
```

is equivalent to the Inductor component declaration from the previous example.

See Also

Related Examples

- “Composite Component — DC Motor” on page 2-95

More About

- “Declaring Member Components” on page 2-59
- “Parameterizing Composite Components” on page 2-60
- “Specifying Component Connections” on page 2-64

Specifying Component Connections

In this section...

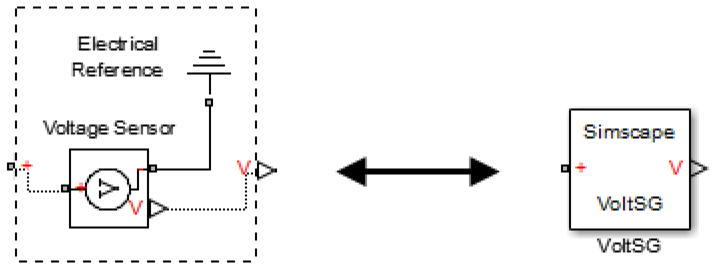
About the Structure Section

The structure section of a Simscape file 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(ExternalAccess=observe)
     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(ExternalAccess=observe)
    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
```

Connections to Implicit Reference Node

The * symbol indicates connections to a reference node in `branch` statements. You can also use it to indicate connections to an implicit reference node within the structure section of a component:

```
connections
    connect(A, *);
end
```

The implicit reference node acts as a virtual grounding component. A node connected to an implicit reference has all its Across variables equal to 0.

The * symbol is not domain-specific, and the same structure section can contain connections to implicit reference in different domains:

```
component abc
    nodes
        M = foundation.hydraulic.hydraulic;
        N = foundation.electrical.electrical;
    end
    connections
        connect(M,*);
        connect(N,*);
    end
end
```

However, multiple ports connected to an implicit reference within the same `connect` statement must all belong to the same domain:

```
connections
    connect(a, b, *);
end
```

The order of ports does not matter. This behavior is consistent with general connection rules for multiple conserving ports.

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:

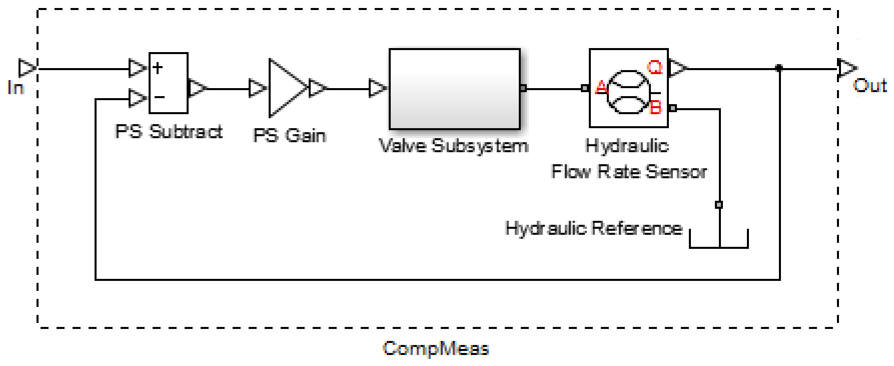
```
  connect(s, d1, d2);
```

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

Source	Destination
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).

Nonscalar Physical Signal Connections

Multidimensional physical signals can be useful for:

- Aggregating measurements at different spatial points, such as temperatures along a coil or a 2-D grid of elements
- Using 3-D body positions or velocities
- Using rotation matrices or quaternions in 3-D
- Using tensors

Simscape language supports nonscalar (vector-valued or matrix-valued) physical signals in `inputs` and `outputs` declarations. All signals in such vector or matrix should have the same units. For example, the following declaration

```
inputs
  I = {zeros(3), 'm/s'}; % :left
end
```

initializes a component input as a 3-by-3 matrix of linear velocities.

When you connect input and output ports carrying nonscalar physical signals, you can use signal indexing and concatenation at the source, but not at the destination. Scalar expansion is not allowed.

The following table shows valid syntax examples, assuming subcomponent A with output signal port `A.o` is being connected to subcomponent B with input signal port `B.i`, and all sizes and units are compatible.

Construct	Explanation
`connect(A.o(1,2), B.i);`	Source indexing, to connect to a scalar destination: take entry (1,2) of the output A.o and connect it to the input B.i.
`connect(A.o(1:2:5,2:3), B.i);`	Index by rows and columns to specify a submatrix.
`connect(A.o(1:2:end,:), B.i);`	Use colon notation to specify array boundaries (pass every other column of the output A.o to input B.i.
`connect([A1.o, A2.o], B.i);`	Concatenate outputs A1.o and A2.o column-wise and pass the result to the input B.i.

You can use block parameter values for indexing inside a `connect` statement, for example:

`connect(a.o(value(`*`param_name`*`, '1'), 3), b.i);`

When you connect two physical signals, their units must be directly convertible. If one of the signals is declared as unitless (that is, with units of `'1'`) , then you can connect a signal with any base units to it. However, unit conversion is not supported in this case. For example, if `a.i` is a 2x1 unitless input port, then this statement is valid:

```
connect([out1_in_meters, out2_in_seconds], a.i);
```

If you connect signals with different scales of the same unit with a unitless input port, the compiler alerts you to the fact that unit conversion is ignored. For example, the following statement produces a warning at compile time:

```
connect([out1_in_km, out2_in_mm], a.i);
```

See Also

Related Examples

- “Composite Component — DC Motor” on page 2-95

More About

- “Declaring Member Components” on page 2-59
- “Parameterizing Composite Components” on page 2-60
- “Specifying Initial Target Values for Member Variables” on page 2-62

Converting Subsystems into Composite Components

In this section...

The `subsystem2ssc` function lets you convert a subsystem consisting entirely of Simscape blocks into a textual Simscape file. The function generates a composite component file based on the subsystem configuration. If the subsystem being converted contains nested subsystems, then the function generates several Simscape files, one for each subsystem.

Use this functionality to:

- Facilitate the authoring of composite components. When writing textual files, it can be difficult to visualize the connections inside a composite component. This functionality lets you create a model out of Simscape blocks, enclose it into a subsystem, and then convert this subsystem into a textual composite component.
- Improve the usability of a complex subsystem, by reducing clutter and exposing only a few relevant parameters at the top level.
- Share your models with customers without revealing the underlying intellectual property.

Suggested Workflows

To create a reusable composite component:

1. Model a physical component (such as a motor, valve, amplifier, and so on) using blocks from the Simscape Foundation library, add-on product libraries, or custom blocks. Fine-tune the parameters and troubleshoot the model, as necessary.
2. Select the blocks and connection lines that represent your physical component, and create a subsystem from selection. For more information, see "Create Subsystems".

   The subsystem does not need to be masked. However, to expose underlying block parameters or variables at the top level, you have to mask the subsystem and promote these parameters or variables to the subsystem mask. For more information, see "Parameter Promotion" on page 2-71.
3. Use the `subsystem2ssc` function to convert your subsystem into a textual composite component. If the subsystem being converted contains nested subsystems, then the function generates several Simscape files, one for each subsystem.

To enable model sharing without revealing the underlying intellectual property:

1. When converting the subsystem, use the `subsystem2ssc` function with a `targetFolder` argument to place the file or files generated by the function into a target folder.

   For example,

   ```
   subsystem2ssc('ssc_dcmotor/DC Motor','./MotorsLibrary')
   ```

   creates a file named `DC_Motor.ssc` and places it into the folder named `MotorsLibrary`.

2. Create and place other motor models into the same target folder.
3. Protect the source files in the target folder by using the `ssc_protect` function.
4. Share the contents of the folder with other users or customers without revealing the underlying source.

You can place generated files into a package folder and build a library by using the `ssc_build` or `ssc_mirror` functions. However, if your subsystem contains nested subsystems, you have to edit the subcomponent paths in the generated files manually to match your intended package structure. Alternatively, you can use the Simscape Component blocks, which work with the flat hierarchy of the target folder without modification.

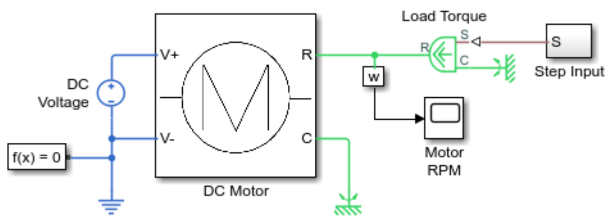
Parameter Promotion

You can mark member block and subsystem parameters for promotion to the top level using the subsystem mask. The `subsystem2ssc` function automatically generates the corresponding Simscape code, similar to composite components. For more information, see “Parameterizing Composite Components” on page 2-60.

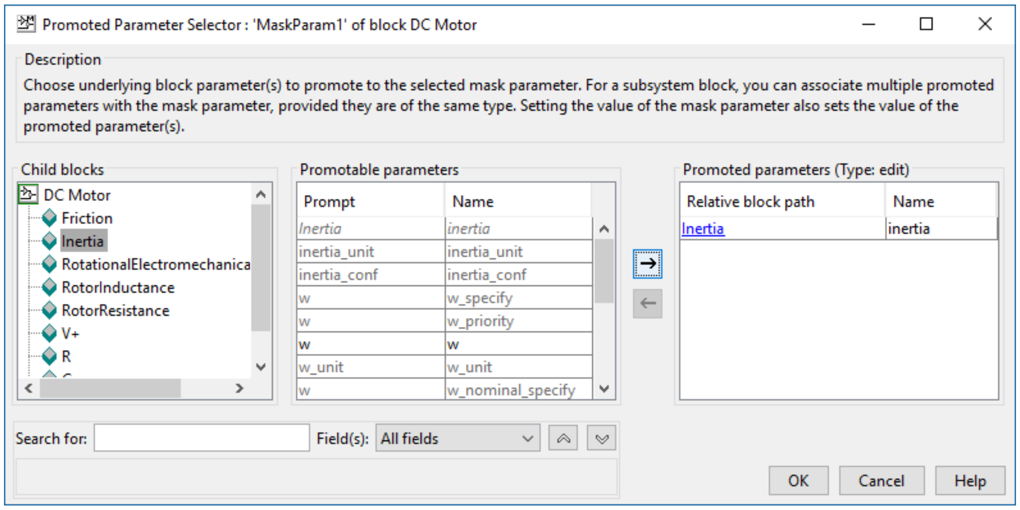
When you deploy the generated composite file as a custom block, the block dialog box contains these promoted parameters only.

This example shows how you can make the motor inertia modifiable at the DC Motor subsystem level, and the effect on generated Simscape code and the resulting custom block mask:

1. Open the Permanent Magnet DC Motor example model by typing `ssc_dcmotor` in the MATLAB Command Window.



2. Right-click the DC Motor subsystem and, from the context menu, select **Mask** > **Edit Mask**.
3. Click the **Parameters & Dialog** tab. Use the **Promote** control option to promote the **Inertia** parameter of the Inertia block to the subsystem mask. For more information, see “Promote Underlying Parameters to Subsystem Mask”.



Promoted Parameter Selector : 'MaskParam1' of block DC Motor

Description

Choose underlying block parameter(s) to promote to the selected mask parameter. For a subsystem block, you can associate multiple promoted parameters with the mask parameter, provided they are of the same type. Setting the value of the mask parameter also sets the value of the promoted parameter(s).

Child blocks
- DC Motor
  - Friction
  - Inertia
  - RotationalElectromechanica
  - RotorInductance
  - RotorResistance
  - V+
  - R

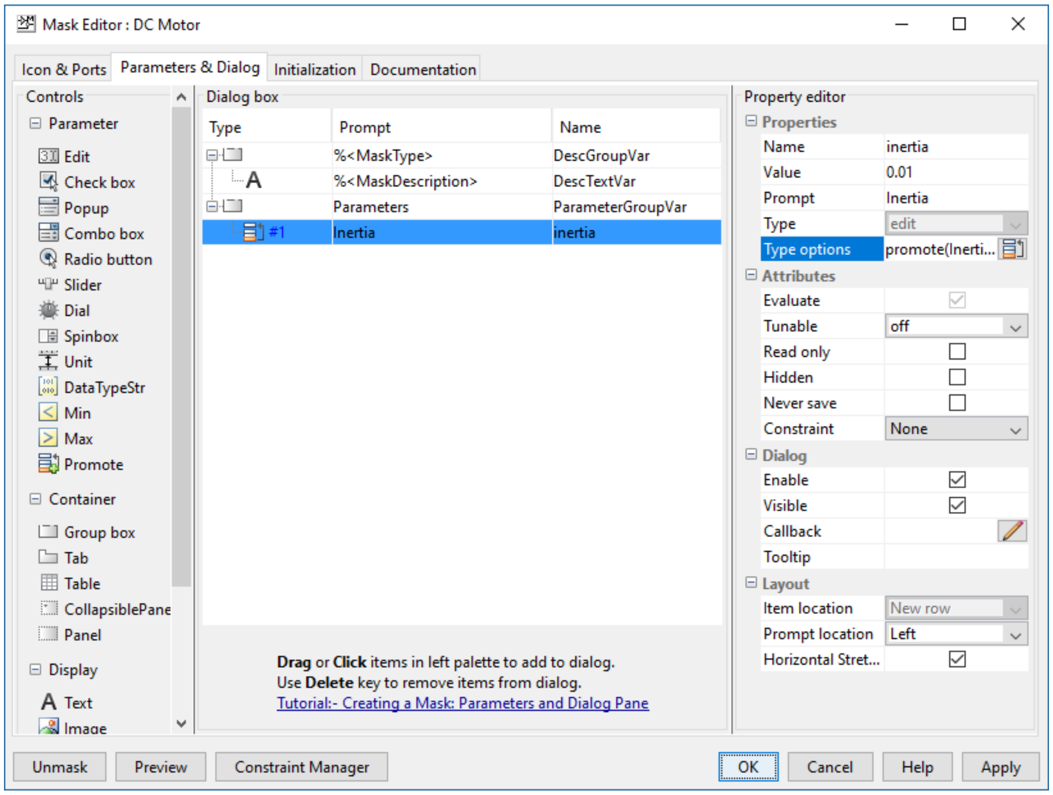
Promotable parameters

Prompt	Name
Inertia	inertia
inertia_unit	inertia_unit
inertia_conf	inertia_conf
w	w_specify
w	w_priority
w	w
w_unit	w_unit
w	w_nominal_specify

Promoted parameters (Type: edit)

Relative block path	Name
Inertia	inertia

Search for: Field(s): All fields



Mask Editor : DC Motor

Controls
- Parameter
  - Edit
  - Check box
  - Popup
  - Combo box
  - Radio button
  - Slider
  - Dial
  - Spinbox
  - Unit
  - DataTypeStr
  - Min
  - Max
  - Promote
- Container
  - Group box
  - Tab
  - Table
  - CollapsiblePane
  - Panel
- Display
  - Text
  - Image

Dialog box

Type	Prompt	Name
	%<MaskType>	DescGroupVar
A	%<MaskDescription>	DescTextVar
	Parameters	ParameterGroupVar
#1	Inertia	inertia

Drag or **Click** items in left palette to add to dialog.
Use **Delete** key to remove items from dialog.
Tutorial:- Creating a Mask: Parameters and Dialog Pane

Property editor

Properties	
Name	inertia
Value	0.01
Prompt	Inertia
Type	edit
Type options	promote(Inerti...
Attributes	
Evaluate	✓
Tunable	off
Read only	☐
Hidden	☐
Never save	☐
Constraint	None
Dialog	
Enable	✓
Visible	✓
Callback	
Tooltip	
Layout	
Item location	New row
Prompt location	Left
Horizontal Stret...	✓

Alternatively, you can use the **Edit** control option to add a parameter to the subsystem mask and associate it with the **Inertia** parameter of the underlying Inertia block.

4 Convert the DC Motor subsystem into a Simscape component file and place this file in your current working folder:

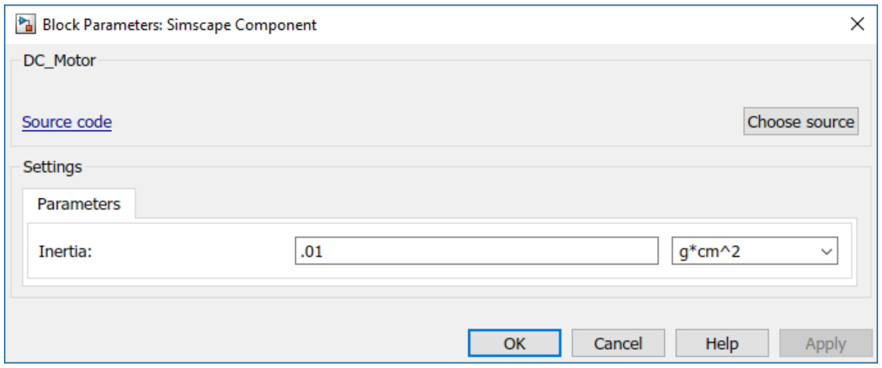
```
subsystem2ssc('ssc_dcmotor/DC Motor')
```

The function creates a file named `DC_Motor.ssc` in the current folder. Open the file in the editor.

```
component DC_Motor
  parameters
    inertia = {.01, 'cm^2*g'}; %Inertia
  end
  nodes
    C = foundation.mechanical.rotational.rotational;
    R = foundation.mechanical.rotational.rotational;
    V1 = foundation.electrical.electrical;
    V0 = foundation.electrical.electrical;
  end
  components(ExternalAccess = observe)
    Rotor_Resistance = foundation.electrical.elements.resistor(R = {3.9, 'Ohm'});
    Rotor_Inductance = foundation.electrical.elements.inductor(l = {1.2e-05, 'H'}, r = {0, 'Ohm'}, g = {1e-09, '1/Ohm'}, i_L = {va
    Rotational_Electromechanical_Converter = foundation.electrical.elements.rotational_converter(K = {.0006875493541569879, 's*V/ra
    Inertia = foundation.mechanical.rotational.inertia(inertia = inertia);
    Friction = foundation.mechanical.rotational.friction(brkwy_trq = {2e-05, 'm*N'}, brkwy_vel = {.03347, 'rad/s'}, Col_trq = {2e-0
  end
  connections
    connect(V0,Rotor_Resistance.p);
    connect(Rotational_Electromechanical_Converter.p,Rotor_Inductance.n);
    connect(V1,Rotational_Electromechanical_Converter.n);
    connect(Rotor_Inductance.p,Rotor_Resistance.n);
    connect(R,Friction.R);
    connect(R,Inertia.I);
    connect(R,Rotational_Electromechanical_Converter.R);
    connect(C,Friction.C);
    connect(C,Rotational_Electromechanical_Converter.C);
  end
end
```

Notice the top-level `parameters` block containing the `inertia` parameter.

5 If you now point a Simscape Component block to the `DC_Motor.ssc` source file, the block dialog box contains a parameter named **Inertia**.


Block Parameters: Simscape Component

DC_Motor

Source code

Settings

Parameters

Inertia: .01 g*cm^2

Limitations

The subsystem being converted must consist entirely of blocks authored in Simscape language, such as blocks from the Simscape Foundation library, add-on product libraries, or custom blocks. Blocks from the Simscape "Utilities" library are not authored in Simscape language, therefore:

- If the subsystem contains a Simscape Component block, then during the conversion this block is replaced by its source component.
- Connection Port blocks are represented by the `connect` statements.
- Other blocks from the Utilities library (Solver Configuration, Simscape Bus, and so on) are not allowed because they have no equivalent textual representation.

The subsystem being converted cannot contain multiple Simscape networks.

If the subsystem being converted contains nested subsystems, you might have to manually edit the references to the generated files for nested subsystems when running `ssc_build` on the package.

If you use blocks from Simscape libraries, keep the original subsystem used to generate the composite component. Simscape language does not support forwarding tables or versioning. As a result, if the underlying library blocks undergo changes in a future release, a textual composite component generated from these blocks might stop working. If that happens, open the original subsystem in the new release and rerun the conversion.

See Also

`components` | `connections` | `ssc_build` | `ssc_mirror` | `ssc_protect` | `subsystem2ssc`

More About

- “Declaring Member Components” on page 2-59
- “Parameterizing Composite Components” on page 2-60
- “Specifying Component Connections” on page 2-64
- “Building Custom Block Libraries” on page 4-25

Defining Component Variants

In this section...

Physical modeling often requires incremental modeling approach. It is a good practice to start with a simple model, run and troubleshoot it, then add the desired special effects, like fluid compressibility or fluid inertia. Another example is modeling a diode with different levels of complexity: linear, zener diode, or exponential. Composite components often require conditional inclusion of a certain member component and a flexible connection scheme.

Including different modeling variants within a single component requires applying control logic to determine the model configuration. You achieve this goal by using conditional sections in a component file.

Conditional Sections

A conditional section is a top-level section guarded by an `if` clause. Conditional sections are parallel to other top-level sections of a component file, such as declaration or equations sections.

A conditional section starts with an `if` keyword and ends with an `end` keyword. It can have optional `elseif` and `else` branches. The body of each branch of a conditional section can contain declaration blocks, equations, structure sections, and so on, but cannot contain the `setup` function.

The `if` and `elseif` branches start with a predicate expression. If a predicate is true, the branch gets activated. When all predicates are false, the `else` branch (if present) gets activated. The compiled model includes elements (such as declarations, equations, and so on) from active branches only.

```
component MyComp
  [...]
  if Predicate1
    [...] % body of branch1
  elseif Predicate2
    [...] % body of branch2
  else
    [...] % body of branch3
  end
  [...]
end
```

Unlike the `if` statements in the equations section, different branches of a conditional section can have different variables, different number of equations, and so on. For example, you can have two variants of a pipe, one that accounts for resistive properties only and the second that also models fluid compressibility:

```
component MyPipe
  parameters
     fl_c = 0; % Model compressibility? (0 - no, 1 - yes)
  end
  [...] % other parameters, variables, branches
  if fl_c == 0
    equations
       % first set of equations, resistive properties only
    end
  else
    variables
       % additional variable declarations, needed to account for fluid compressibility
    end
    equations
       % second set of equations, including fluid compressibility
    end
  end
end
```

In this example, if the block parameter **Model compressibility? (0 - no, 1 - yes)** is set to 0, the first set of equations gets activated and the block models only the resistive properties of the pipe. If the block user changes the value of the parameter, then the `else` branch gets activated, and the compiled model includes the additional variables and equations that account for fluid compressibility.

Note Enumerations are very useful in defining component variants, because they let you specify a discrete set of acceptable parameter values. For an example of how this component can use enumeration, see "Using Enumeration in Predicates" on page 3-18.

Rules and Restrictions

Nested conditional sections are allowed. For example:

```
component A
  parameters
     p1 = 0;
     p2 = 0;
     p3 = 0;
  end
  if p1 > 0
   [...]
     if p2 > 0
        [...]
     end
     if p3 > 0
        [...]
     end
   [...]
   end
end
```

Predicates must be parametric expressions, because the structure of a model must be fixed at compile time and cannot change once the model is compiled. Using a variable in a predicate results in a compile-time error:

```
component A
   [...]
   variables
     v = 0;
   end
   if v > 0  % error: v>0 is not a parametric expression because v is a variable
     [...]
   else
     [...]
   end
 end
```

Predicates may depend on parameters of the parent (enclosing) component. They may not depend, directly or indirectly, on parameters of member (embedded) components or on domain parameters:

```
component A
   parameters
     p = 1;
   end
   parameters(Access=private)
     pp = c.p;
   end
   components
     c = MyComp;
   end
   nodes
     n = MyDomain;
   end
   if p > 0  % ok
     [...]
   elseif c.p > 0 % error: may not depend on parameters of embedded component
     [...]
   elseif n.p > 0 % error: may not depend on domain parameters
     [...]
   elseif pp > 0 % error: pp depends on c.p
     [...]
   end
 end
```

Accessibility of class members declared inside conditional sections is equivalent to private class members (`Access=private`). They are not accessible from outside the component class, even if their branch is active.

The scope of the class members declared inside a conditional section is the entire component class. For example:

```
component A
   nodes
     p = foundation.electrical.electrical;
     n = foundation.electrical.electrical;
   end
   parameters
     p1 = 1;
   end
   if p1 > 0
     components
       r1 = MyComponentVariant1;
     end
   else
     components
       r1 = MyComponentVariant2;
     end
   end
   connections
     connect(p, r1.p);
     connect(n, r1.n);
   end
 end
```

However, using a conditional member outside the conditional section when the branch is not active results in a compilation error:

```
component A
   nodes
     p = foundation.electrical.electrical;
     n = foundation.electrical.electrical;
   end
   parameters
     p1 = 0;
   end
   if p1 > 0
     components
       r1 = MyComponentVariant1;
     end
   end
   connections
     connect(p, r1.p); % error if p1=0 and the predicate is false
   end
 end
```

Parameters that are referenced by predicates of conditional sections, directly and indirectly, must be compile-time parameters. The `setup` function may not write to these parameters, for example:

```
component A
   parameters
     p1 = 1;
   end
   if p1 > 0  % p1 is a compile-time parameter
     [...]
   else
     [...]
   end
   function setup
     tmp = p1; % ok to read from p1
     p1 = 10;  % error: may not write to p1 here
   end
 end
```

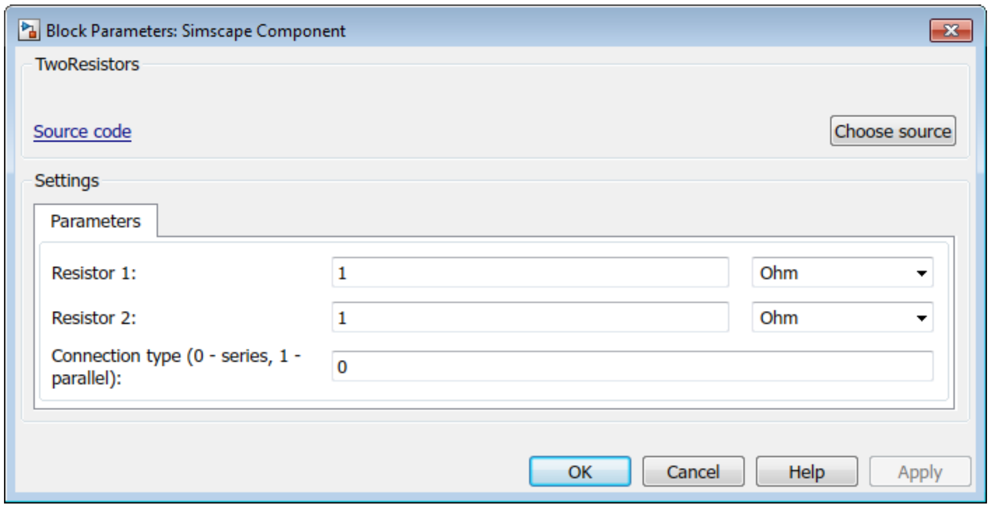
Example

This simple example shows a component containing two resistors. The resistors can be connected either in series or in parallel, depending on the value of the control parameter:

```
component TwoResistors
  nodes
     p = foundation.electrical.electrical; % +:left
     n = foundation.electrical.electrical; % -:right
  end
  parameters
    p1 = {1, 'Ohm'};   % Resistor 1
    p2 = {1, 'Ohm'};   % Resistor 2
    ct = 0;            % Connection type (0 - series, 1 - parallel)
  end
  components(ExternalAccess=observe)
    r1 = foundation.electrical.elements.resistor(R=p1);
    r2 = foundation.electrical.elements.resistor(R=p2);
  end
  if ct == 0      % linear connection
    connections
      connect(p, r1.p);
      connect(r1.n, r2.p);
      connect(r2.n, n);
    end
  else          % parallel connection
```

```
    connections
      connect(r1.p, r2.p, p);
      connect(r1.n, r2.n, n);
    end
  end
end
```

To test the correct behavior of the conditional section, point a Simscape Component block to this component file. Place the block in a circuit with a 10V DC voltage source and a current sensor. With the default parameter values, the resistors are connected in series, and the current is 5A.


Block Parameters: Simscape Component

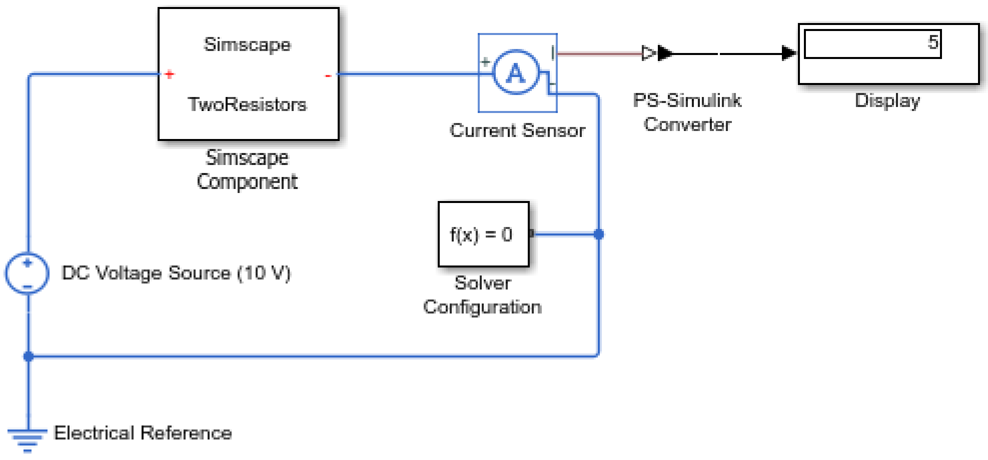
TwoResistors

Source code

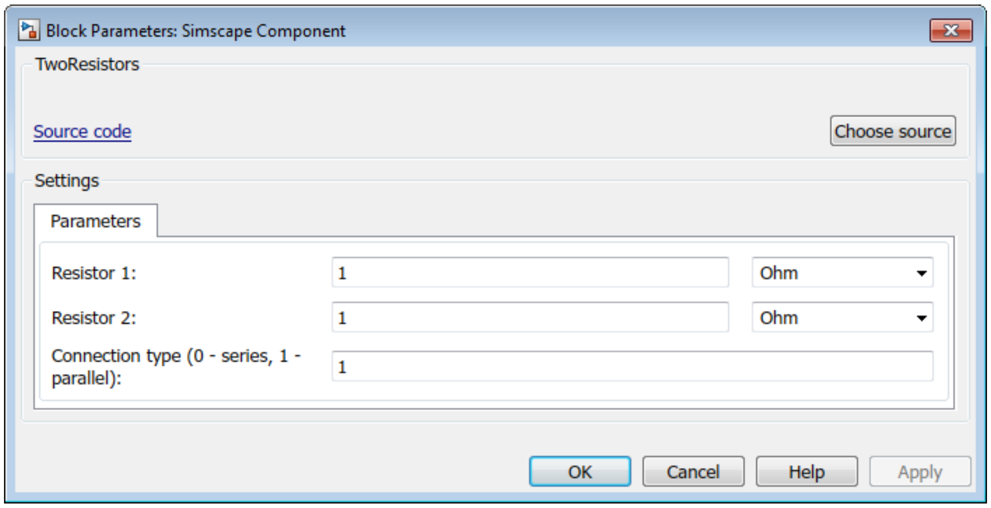
Settings

Parameters

Resistor 1:	1	Ohm
Resistor 2:	1	Ohm
Connection type (0 - series, 1 - parallel):	0	



If you change the value of the **Connection type (0 - series, 1 - parallel)** parameter to 1, the resistors are connected in parallel, and the current is 20A.


Block Parameters: Simscape Component

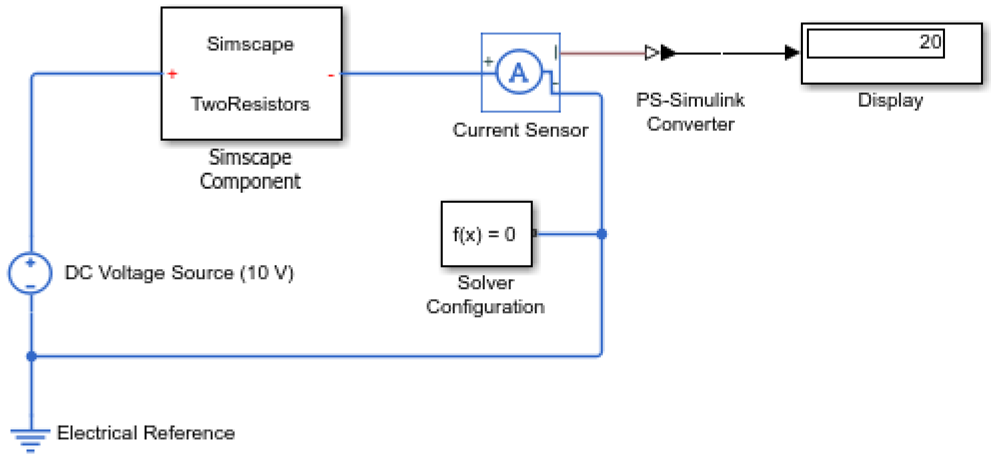
TwoResistors

Source code

Settings

Parameters

Resistor 1:	1	Ohm
Resistor 2:	1	Ohm
Connection type (0 - series, 1 - parallel):	1	



See Also

More About

- “Defining Conditional Visibility of Component Members” on page 2-81
- “Component Variants — Series RLC Branch” on page 2-83
- “Component Variants — Thermal Resistor” on page 2-85

Defining Conditional Visibility of Component Members

The `annotations` section in a component file lets you control visibility of component members, such as parameters and nodes, in block icons and dialog boxes. When you declare a component member, the `ExternalAccess` attribute sets the visibility of the member in the user interface, that is, in block dialog boxes, simulation logs, variable viewer, and so on. The `annotations` section serves a similar purpose, but it is especially useful for block variants because it lets you define conditional visibility of component members based on a predicate condition.

When you define component variants using conditional declarations, certain parameters, variables, or ports can be used in one block variant but not in others. For example, you have a component that models hydraulic pipelines with circular and noncircular cross sections. For a circular pipe, you need to specify its internal diameter. For a noncircular pipe, you need to specify its hydraulic diameter and pipe cross-sectional area. You can now use the `annotations` section to control the visibility of these parameters in the block dialog box:

```
component MyPipe
  parameters
    circular  = true;             % Circular pipe?
    d_in      = { 0.01, 'm' };    % Pipe internal diameter
    area      = { 1e-4, 'm^2' };  % Noncircular pipe cross-sectional area
    D_h       = { 1.12e-2, 'm' }; % Noncircular pipe hydraulic diameter
  end
  if circular
  % Hide inapplicable parameters
    annotations
       [area, D_h] : ExternalAccess=none;
    end
    equations
       % first set of equations, for circular pipe
    end
  else
  % Hide inapplicable parameter
    annotations
       d_in : ExternalAccess=none;
    end
    equations
       % second set of equations, for noncircular pipe
    end
  end
  [...] % other parameters, variables, branches, equations
end
```

Similar to other types of conditional declarations, a predicate of a conditional annotation must be a parametric expression that evaluates to true or false. However, there is an additional restriction that all the parameters used in the predicate of a conditional annotation must be either of type logical or enumerated. In this example, the `circular` parameter is of type logical.

The `annotations` section lets you control visibility of the following component members:

- Parameters
- Variables
- Nodes
- Inputs
- Outputs

The `annotations` section also lets you specify conditional custom icons. This is especially useful if the number of ports changes for different variants. For example:

```
component MyPipe
  parameters
```

```
    thermal_variant = false; % Model thermal effects?
  end
  if thermal_variant
  % Use icon with additional thermal port
    annotations
       Icon = 'pipe_thermal.jpg';
    end
  else
  % Use regular icon, with two fluid ports
    annotations
       Icon = 'pipe.jpg';
    end
  end
  [...] % Other parameters, variables, nodes, branches, equations
end
```

For more information on using custom block icons, see “Customize the Block Icon” on page 4-42.

Rules and Restrictions

The predicate of a conditional annotation must be a parametric expression that evaluates to true or false. All the parameters used in the predicate of a conditional annotation must be either of type logical or enumerated.

Member attributes must be uniquely defined, which means that the same member cannot be declared more than once, with different values of the same attribute. The only exception to this rule is the use of `ExternalAccess` attribute in the `annotations` section. You can declare a component member with a certain value of `ExternalAccess`, and then specify a different `ExternalAccess` attribute value in the `annotations` section, for example:

```
component MyPipe
  parameters
    circular  = true;            % Circular pipe?
  end
  parameters(ExternalAccess=none)
    d_in      = { 0.01, 'm' };    % Pipe internal diameter
  [...]
  end
  if circular
  % Expose pipe diameter
    annotations
       d_in : ExternalAccess=modify;
    end
  [...]
```

In case of conflict, the `ExternalAccess` attribute value specified in the `annotations` section overrides the value specified for that member in the declaration section. For a complete component example using this approach, see “Component Variants — Thermal Resistor” on page 2-85.

See Also

More About

- “Defining Component Variants” on page 2-75
- “Component Variants — Thermal Resistor” on page 2-85
- “Component Variants — Series RLC Branch” on page 2-83

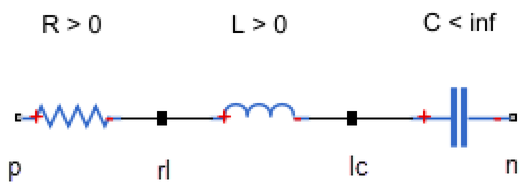
Component Variants — Series RLC Branch

The following example shows a series RLC component that implements a single resistor, inductor, or capacitor, or a series combination of these elements. The component uses conditional sections to implement the control logic.

```
import foundation.electrical.electrical;   % electrical domain class definition
import foundation.electrical.elements.*;   % electrical elements
component SeriesRLC
   nodes
     p = electrical; % +:left
     n = electrical; % -:right
   end
   nodes(Access=protected, ExternalAccess=none)
     rl = electrical; % internal node between r and l
     lc = electrical; % internal node between l and c
   end
   parameters
     R = {0, 'Ohm'};
     L = {0, 'H'};
     C = {inf, 'F'};
   end
   if R > 0
     components
       r = resistor(R=R);
     end
     connections
       connect(p, r.p);
       connect(r.n, rl);
     end
   else
     connections
       connect(p, rl); % short circuit p--rl
     end
   end
   if L > 0
     components
       l = inductor(l=L);
     end
     connections
       connect(rl, l.p);
       connect(l.n, lc);
     end
   else
     connections
       connect(rl, lc); % short circuit rl--lc
     end
   end
   if value(C, 'F') < inf
     components
       c = capacitor(c=C);
     end
     connections
       connect(lc, c.p);
       connect(c.n, n);
     end
   else
```

```
      connections
        connect(lc, n); % short circuit lc--n
      end
    end
  end
```

The **R**, **L**, and **C** parameters are initialized to `0`, `0`, and `inf`, respectively. If the block user specifies a nonzero resistance, nonzero impedance, or finite capacitance, the appropriate branch gets activated. The active branch declares the appropriate member component and connects it in series. Each of the `else` clauses short-circuits the appropriate nodes.



Internal nodes `rl` and `lc`, which serve to connect the member components, should not be accessible from outside the component. Set their `Access` attribute to `protected` or `private`. Their `ExternalAccess` attribute is `none`, so that these nodes are not visible on the block icon.

See Also

More About

- “Defining Component Variants” on page 2-75
- “Parameterizing Composite Components” on page 2-60
- “Specifying Component Connections” on page 2-64

Component Variants — Thermal Resistor

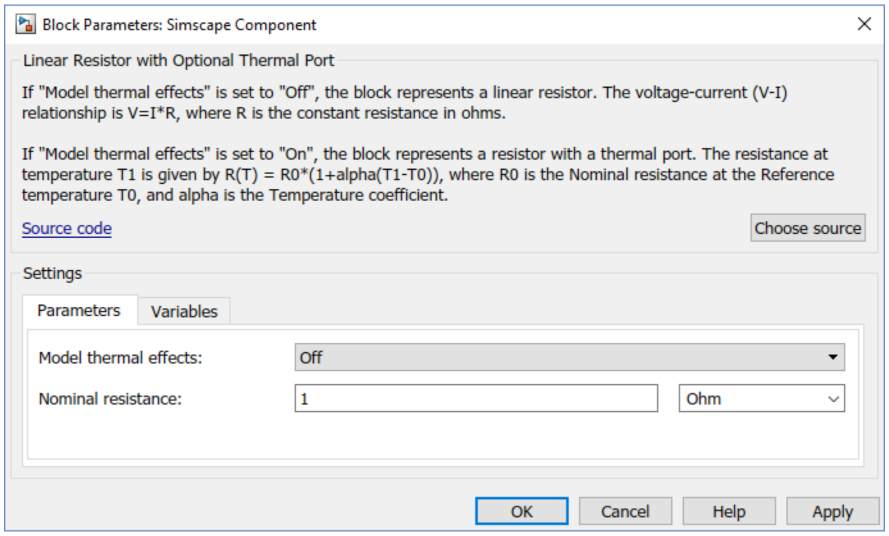
The following example shows a linear resistor with an optional thermal port. The component uses conditional sections to implement the control logic. The `annotations` sections within the conditional branches selectively expose or hide appropriate ports, parameters, and variables based on the value of the control parameter. The two block variants have a different number of ports, and therefore the custom block icon also changes accordingly.

```
component CondResistor
% Linear Resistor with Optional Thermal Port
% If "Model thermal effects" is set to "Off", the block represents a
% linear resistor. The voltage-current (V-I) relationship is V=I*R,
% where R is the constant resistance in ohms.
%
% If "Model thermal effects" is set to "On", the block represents a
% resistor with a thermal port. The resistance at temperature T1 is given by
% R(T) = R0*(1+alpha(T1-T0)), where R0 is the Nominal resistance at the
% Reference temperature T0, and alpha is the Temperature coefficient.

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

parameters
    thermal_effects = simscape.enum.onoff.off; % Model thermal effects
end

parameters(ExternalAccess=none)
    R = { 1, 'Ohm' };         % Nominal resistance
    T0 = {300,'K'};           % Reference temperature
    alpha = {50e-6,'1/K'};    % Temperature coefficient
    tc = {10,'s'};            % Thermal time constant
    K_d = {1e-3,'W/K'};       % Dissipation factor
end

variables(ExternalAccess=none)
    i = { 0, 'A' };                                      % Current
    v = { 0, 'V' };                                      % Voltage
    T1 = {value = {300,'K'}, priority = priority.high};  % Temperature
end

branches
    i : p.i -> n.i;
end

equations
    v == p.v - n.v;
end

if thermal_effects == simscape.enum.onoff.off
    annotations
        % Show non-thermal settings
        Icon = 'custom_resistor.png';
        [R, i, v] : ExternalAccess=modify;
        % Hide thermal node
```

```
            H : ExternalAccess=none;
        end
        connections
            connect(H, *); % Connect hidden thermal node to reference
        end
        equations
            R*i == v;
            T1 == T0;    % Temperature is constant
        end

    else
        annotations
            % Show thermal settings
            Icon = 'custom_resistor_thermal.png';
            [T1, T0, alpha, tc, K_d, H] : ExternalAccess=modify;
        end

        % Add heat flow + thermal equations
        variables(Access=private)
            Q = { 0, 'J/s' }; % Heat flow
        end
        branches
            Q : H.Q -> *
        end
        equations
            T1 == H.T;
            let
                mc = tc*K_d; % mc in Q = m*c*dT
                % Calculate R(T), protecting against negative values
                Rdem = R*(1+alpha*(T1-T0));
                R_T = if Rdem > 0, Rdem else {0,'Ohm'} end;
            in
                R_T*i == v; % Electrical equation
                mc * T1.der == Q + R_T*i*i; % Thermal equation
            end
        end

    end
    end
```

The component initially declares all the optional parameters and variables with the `ExternalAccess` attribute set to `none`, and then exposes them selectively by using the conditional `annotations` sections. The opposite method, of hiding inapplicable members, is also valid, but this approach is more easily scalable when you have multiple component configurations.

If the control parameter, **Model thermal effects**, is set to `Off`, the block represents a linear resistor. The only exposed block parameter is **Nominal resistance**, the **Variables** tab lets you set targets for **Current** and **Voltage**, and the block icon has two ports, **+** and **-**.


Block Parameters: Simscape Component

Linear Resistor with Optional Thermal Port

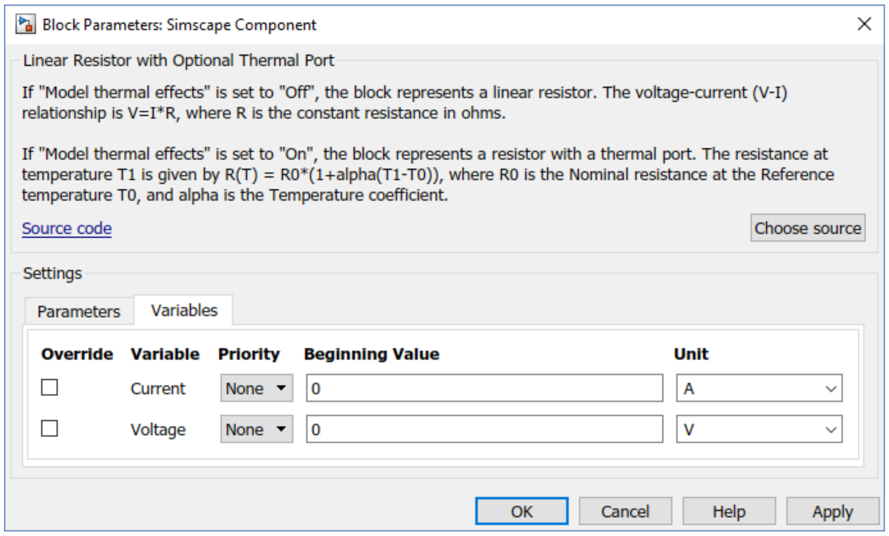
If "Model thermal effects" is set to "Off", the block represents a linear resistor. The voltage-current (V-I) relationship is V=I*R, where R is the constant resistance in ohms.

If "Model thermal effects" is set to "On", the block represents a resistor with a thermal port. The resistance at temperature T1 is given by R(T) = R0*(1+alpha(T1-T0)), where R0 is the Nominal resistance at the Reference temperature T0, and alpha is the Temperature coefficient.

Source code

Settings

Model thermal effects:	Off	
Nominal resistance:	1	Ohm


Block Parameters: Simscape Component

Linear Resistor with Optional Thermal Port

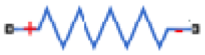
If "Model thermal effects" is set to "Off", the block represents a linear resistor. The voltage-current (V-I) relationship is V=I*R, where R is the constant resistance in ohms.

If "Model thermal effects" is set to "On", the block represents a resistor with a thermal port. The resistance at temperature T1 is given by R(T) = R0*(1+alpha(T1-T0)), where R0 is the Nominal resistance at the Reference temperature T0, and alpha is the Temperature coefficient.

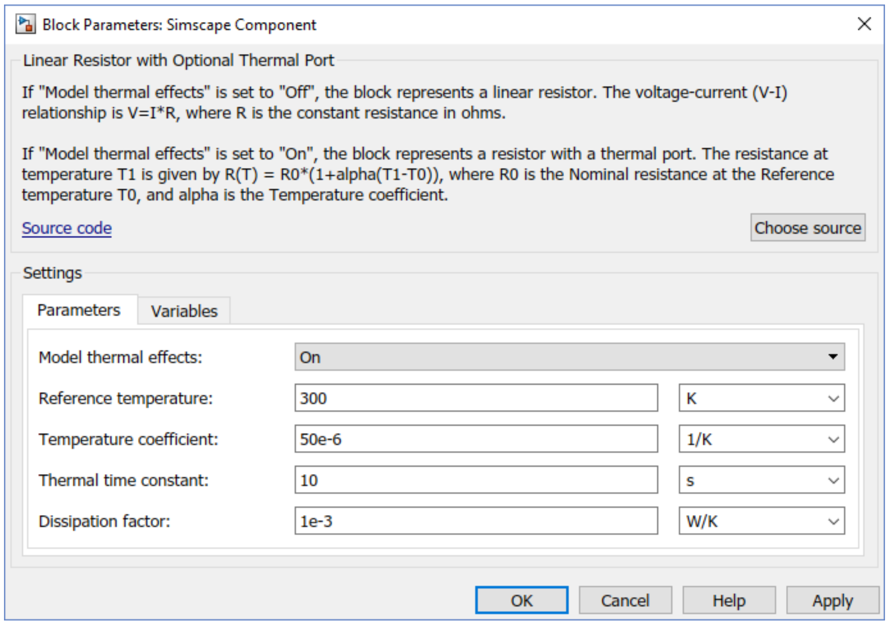
Source code

Settings

Override	Variable	Priority	Beginning Value	Unit
☐	Current	None	0	A
☐	Voltage	None	0	V



If the **Model thermal effects** parameter is set to On, the block represents a resistor with a thermal port, with temperature-dependent resistance. The block parameters, variables, ports, and the custom block icons change accordingly.



Block Parameters: Simscape Component

Linear Resistor with Optional Thermal Port

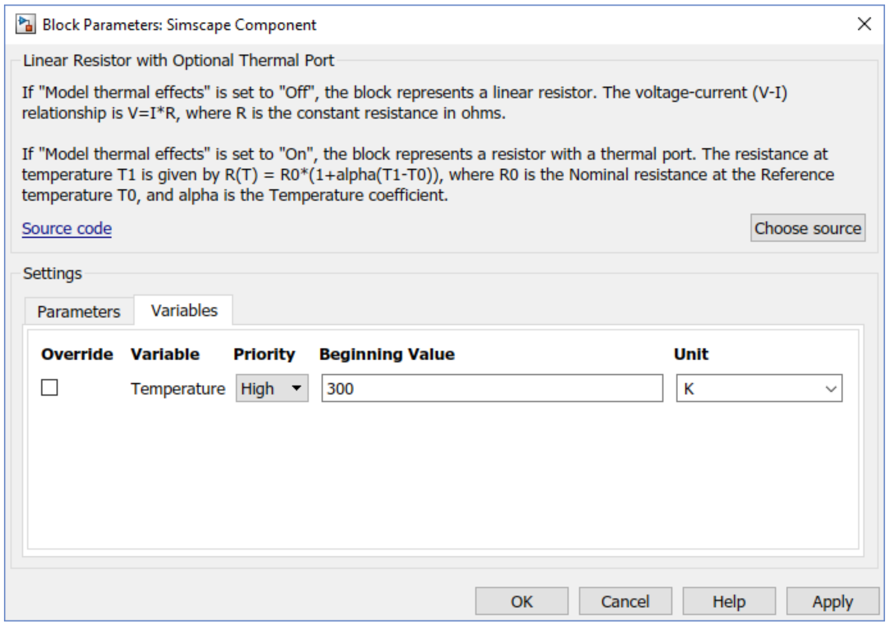
If "Model thermal effects" is set to "Off", the block represents a linear resistor. The voltage-current (V-I) relationship is V=I*R, where R is the constant resistance in ohms.

If "Model thermal effects" is set to "On", the block represents a resistor with a thermal port. The resistance at temperature T1 is given by R(T) = R0*(1+alpha(T1-T0)), where R0 is the Nominal resistance at the Reference temperature T0, and alpha is the Temperature coefficient.

Source code

Settings

Model thermal effects:	On	
Reference temperature:	300	K
Temperature coefficient:	50e-6	1/K
Thermal time constant:	10	s
Dissipation factor:	1e-3	W/K



Block Parameters: Simscape Component

Linear Resistor with Optional Thermal Port

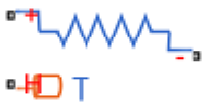
If "Model thermal effects" is set to "Off", the block represents a linear resistor. The voltage-current (V-I) relationship is V=I*R, where R is the constant resistance in ohms.

If "Model thermal effects" is set to "On", the block represents a resistor with a thermal port. The resistance at temperature T1 is given by R(T) = R0*(1+alpha(T1-T0)), where R0 is the Nominal resistance at the Reference temperature T0, and alpha is the Temperature coefficient.

Source code

Settings

Override	Variable	Priority	Beginning Value	Unit
☐	Temperature	High	300	K



See Also

More About

- “Defining Component Variants” on page 2-75
- “Defining Conditional Visibility of Component Members” on page 2-81

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 Through and Across variables later in the file
- Internal variable `theta`, with a default value of `0 rad`, specifying relative angle, that is, deformation of the spring

The `branches` section establishes the relationship between the component Through variable and the component nodes (and therefore the domain Through variable). The `t : r.t -> c.t` statement indicates that the torque through the spring acts from node `r` to node `c`.

The equation section starts with an `assert` construct, which checks that the spring rate is greater than zero. If the block parameter is set incorrectly, the `assert` triggers a run-time error.

The first equation, `w == r.w - c.w`, establishes the relationship between the component Across variable and the component nodes (and therefore the domain Across variable). It defines the angular velocity across the spring as the difference between the node angular velocities.

The following two equations define the spring action:

- `t = k * theta`, that is, torque equals spring deformation times spring rate
- `w = theta.der`, 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
  branches
    t : r.t -> c.t; % torque through from node r to node c
  end
  equations
    assert(k>0)     % spring rate must be greater than zero
    w == r.w - c.w; % velocity across between node r and node c
    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.
- One parameter, `C`, with a default value of `1 F`, specifying the capacitance.
- Through and Across variables, current `i` and voltage `v`, to be connected to the electrical domain Through and Across variables later in the file.

  Variable `v` is declared with high initialization priority, to ensure the initial voltage of `0 V`.

The `branches` section establishes the relationship between the component Through variable and the component nodes (and therefore the domain Through variable). The `i : p.i -> n.i` statement indicates that the current through the capacitor flows from node `p` to node `n`.

The equation section starts with an `assert` construct, which checks that the capacitance value is greater than zero. If the block parameter is set incorrectly, the `assert` triggers a run-time error.

The first equation, `v == p.v - n.v`, establishes the relationship between the component Across variable and the component nodes (and therefore the domain Across variable). It defines the voltage across the capacitor as the difference between the node voltages.

The second equation 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
  end

  variables
    i = { 0, 'A' }; % Current
    v = {value = { 0, 'V' }, priority = priority.high}; % Voltage
  end

  branches
    i : p.i -> n.i; % Current through from node p to node n
  end

  equations
    assert(C > 0)
    v == p.v - n.v; % Voltage across between node p and node n
    i == C*v.der;   % Equation defining the capacitor action
  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 branches 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 later in the file

Note that a Through variable (current) is not declared, and there is no `branches` section.

In the equation section, the first equation, `v == p.v - n.v`, establishes the relationship between the component Across variable, voltage `v1`, and the component nodes (and therefore the domain Across variable at these nodes). It defines the voltage across the sensor as the difference between the node voltages.

The second equation 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), use a branches section, or
% create an equation statement for current (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

  equations
    v1 == p.v - n.v;
    V == v1;
  end

end
```

Grounding Component — Electrical Reference

The easiest way to implement a grounding component is to use a connection to an implicit reference node. For an example of a component that provides an electrical ground to a circuit, see the source for the Electrical Reference block in the Foundation library:

```
component reference
% Electrical Reference :0.5
% Electrical reference port. A model must contain at least one
% electrical reference port (electrical ground).

% Copyright 2005-2016 The MathWorks, Inc.

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

connections
    connect(V, *);
end

end
```

For more information on component connections and the implicit reference node syntax, see “Connections to Implicit Reference Node” on page 2-66.

The following file, `elec_reference.ssc`, shows how to implement a behavioral model of an electrical reference. This component has one node, where the voltage equals zero. It also declares a current variable, makes it incident to the component node using the `branches` section, 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 later in the file. Note that there is no need to declare an Across variable (voltage) because this is a grounding component.

The `branches` section establishes the relationship between the component Through variable, current `i`, and the component nodes (and therefore the domain Through variable). The `i : V.i -> *` statement indicates that the current flows from node `V` to the reference node, indicated as `*`.

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

  branches
    i : V.i -> *;
  end

  equations
    V.v == 0;
  end

end
```

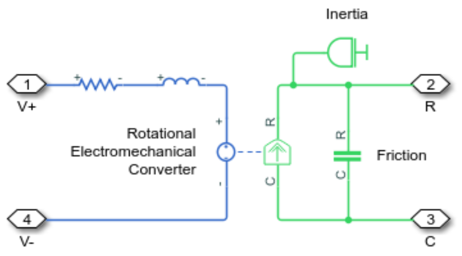
See Also

More About

- “Specifying Component Connections” on page 2-64

Composite Component — DC Motor

In the “Permanent Magnet DC Motor” example, the DC Motor block is implemented as a masked subsystem.

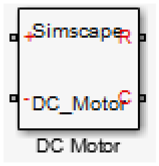


The following code implements the same model by means of a composite component, called `DC Motor`. The composite component uses the components from the Simscape Foundation library as building blocks, and connects them as shown in the preceding block diagram.

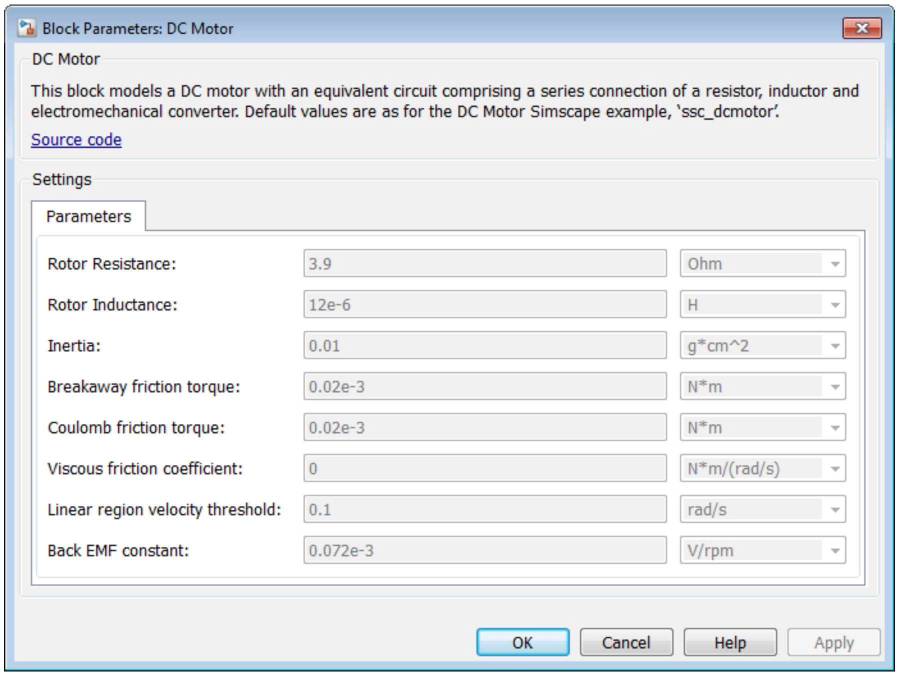
```
component DC_Motor
% DC Motor
% This block models a DC motor with an equivalent circuit comprising a
% series connection of a resistor, inductor, and electromechanical converter.
% Default values are as for the DC Motor Simscape example, ssc_dcmotor.

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
    motor_inertia       = { 0.01, 'g*cm^2' };       % Inertia
    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
    breakaway_velocity  = { 0.1, 'rad/s' };         % Breakaway friction velocity
    back_emf_constant   = { 0.072e-3, 'V/rpm' };    % Back EMF constant
end

components(ExternalAccess=observe)
    rotorResistor                 = foundation.electrical.elements.resistor(R = rotor_resistance);
    rotorInductor                 = foundation.electrical.elements.inductor(l = rotor_inductance);
    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, brkwy_vel = breakaway_velocity);
    motorInertia                  = foundation.mechanical.rotational.inertia(inertia = motor_inertia);
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

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.


Block Parameters: 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'.

Source code

Settings

Parameters

Rotor Resistance:	3.9	Ohm
Rotor Inductance:	12e-6	H
Inertia:	0.01	g*cm^2
Breakaway friction torque:	0.02e-3	N*m
Coulomb friction torque:	0.02e-3	N*m
Viscous friction coefficient:	0	N*m/(rad/s)
Linear region velocity threshold:	0.1	rad/s
Back EMF constant:	0.072e-3	V/rpm

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.

See Also

More About

- “About Composite Components” on page 2-58
- “Declaring Member Components” on page 2-59
- “Parameterizing Composite Components” on page 2-60
- “Specifying Initial Target Values for Member Variables” on page 2-62
- “Specifying Component Connections” on page 2-64

Working with Domain Parameters

In this section...

Declaring Domain Parameters

Similar to a component parameter, you declare each domain parameter as a value with unit on page 2-5. 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.

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 provide the domain parameter values used in the larger model, there are some that simply propagate the parameters, and there are some that do not propagate parameters.

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 2-100.

Source Components

Source components provide a way to modify the domain parameter values. You declare a component parameter, and then use direct assignment to a domain parameter in the component node declaration. This assignment establishes the connection, which lets the parameter of the source component control the domain parameter value.

The following is an example of a source component, connected to the hydraulic domain `t_hyd`, defined in “Propagation of Domain Parameters” on page 2-98. This component provides the value of the temperature parameter to the rest of the model.

```
component  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=t); % t_hyd node with direct parameter assignment
  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. You cannot have more than one block controlling the same domain parameter connected to a circuit, unless different segments of the circuit are separated by a blocking component.

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 equations and other 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 2-98. 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 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`. If your model requires different values of a domain parameter in different segments of the same circuit, use blocking components to separate these segments and connect each segment to its own source component. For more information, see “Attribute Lists” on page 2-103.

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 4-25.
- Build a custom block library based on these Simscape files. For more information, see “Converting Your Simscape Files” on page 4-26.
- 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 2-98.

```
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 2-98. This component provides the value of the temperature parameter to the rest of the model.

```
component  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=t); % t_hyd node with direct parameter assignment
  end
end
```

4. Create the component file `h_temp_sensor.ssc`, as described in “Propagating Components” on page 2-99. 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 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`):

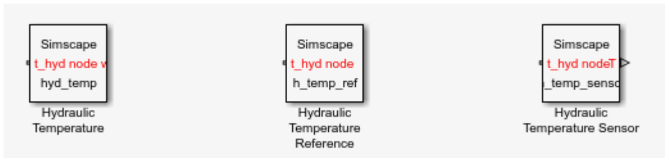
```
component h_temp_ref
% Hydraulic Temperature Reference
%      Provide reference for thermohydraulic circuits
  nodes
    a = THyd.t_hyd; % t_hyd node
  end
  connections
    connect(a, *);
  end
end
```

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:

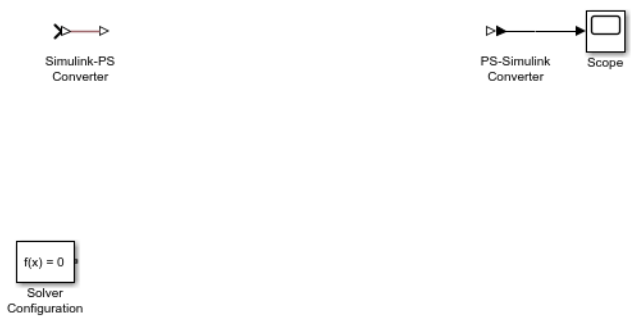
```
THyd_lib
```



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

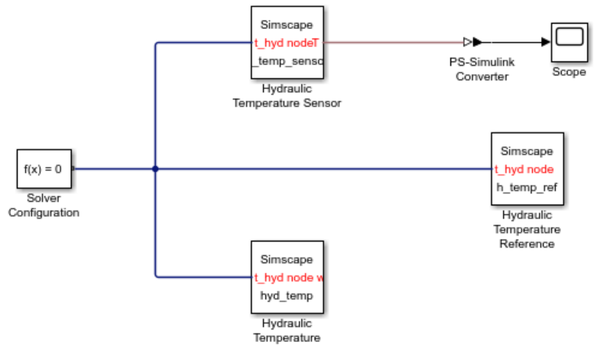
```
ssc_new
```

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



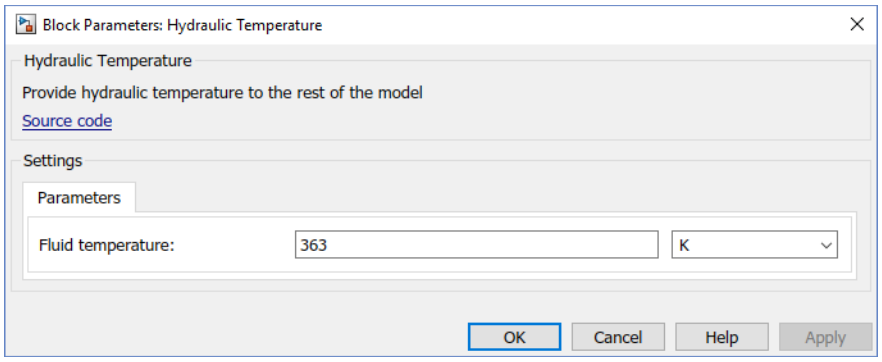
9 Delete the Simulink-PS Converter block, because our model is not going to have any Simulink input signals.

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.


Block Parameters: Hydraulic Temperature

Hydraulic Temperature

Provide hydraulic temperature to the rest of the model

Source code

Settings

Parameters

Fluid temperature: 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

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`
	`blocks`
	`source` (not recommended)
		`propagates`	`component`	Defines the domain data propagation of the component. By default, components propagate domain data, such as domain parameter values. If your model requires different values of a domain parameter in different segments of the same circuit, use `blocks` to designate a blocking component.
				
				Using the `source` value, along with the `setup` function, is no longer recommended; instead, use direct assignment to a domain parameter in the component node declaration. See "Working with Domain Parameters" on page 2-98.

`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 = blocks) Separator
  % component model goes here
end
```

Here, `Propagation` is a model attribute.

Member Attributes

Member attributes apply to a whole declaration block.

Attribute	Values	Default	Member Classes	Description

`Access`	`public`
	`private`
	`protected`
		`public`	all	Defines the read and write access of members. Public (the default) is the most permissive access level. There are no restrictions on accessing public members. Private members are only accessible to the instance of the component model and not to external clients. Protected members of a base class are accessible only to subclasses.

`ExternalAccess`	`modify`
	`observe`
	`none`
		Depends on the value of `Access` attribute: for `public`, the default is `modify`, for `private` and `protected`, the default is `observe`	all	Sets the visibility of the member in the user interface, that is, in block dialog boxes, simulation logs, variable viewer, and so on:
				• `modify` — The member is modifiable in the block dialogs and visible in the logs and viewer.
				• `observe` — The member is visible in the logs and viewer, but not modifiable, and therefore not visible, in block dialogs.
				• `none` — The member is visible nowhere outside the language.

`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 “Declare Through and Across Variables for a Domain” on page 2-6.
`Event`	`true` `false`	`false`	`variables`	If set to `true`, declares event variables for a component. You can set this attribute to `true` only for model type `component`. See “Event Variables” on page 2-52.
`Conversion`	`absolute` `relative`	`absolute`	`parameters` `variables`	Defines how the parameter or variable units are converted for use in equations, intermediates, and other sections. See “Parameter Units” on page 2-11.

Attribute	Values	Default	Member Classes	Description
`MATLABEvaluation`	`default` `compiletime`	`default`	`parameters` `variables`	If a member declaration contains a declaration function that does not support code generation, set this attribute to `compiletime`. The declaration function is then evaluated only at compile time, and all the function input parameters are marked as compile-time only. See “Declaration Functions” on page 3-22.

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

```
parameters (Access = public,ExternalAccess = observe)
  % parameters go here
end
```

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

Specifying Member Accessibility

The two attributes defining member accessibility act in conjunction. The default value of the `ExternalAccess` attribute for a member depends on the value of the `Access` attribute for that member.

`Access`	Default `ExternalAccess`
`public`	`modify`
`protected`	`observe`
`private`	`observe`

You can modify the values of the two attributes independently from each other. However, certain combinations are prohibited. The compiler enforces the following rules:

- Members in the base class with `Access=private` are forced to have `ExternalAccess=none`, to avoid potential collision of names between the base class and the derived class.
- When `Access` is explicitly set to `private` or `protected`, it does not make sense to explicitly set `ExternalAccess=modify` . In this situation, the compiler issues a warning and remaps `ExternalAccess` to `observe`.

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 events, equation, structure, and other sections 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

- Starting in R2019a, using `setup` is not recommended. Other constructs available in Simscape language let you achieve the same results without compromising run-time capabilities. For more information, see “setup is not recommended” on page 5-69.

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, and relationship between the component variables and nodes (and therefore, connects the component variables with the Through and Across domain variables). Such a component is not very useful as a library block, so if you do not want the base class to appear as a block in a custom library, set the `Hidden=true` attribute value:

```
component (Hidden=true) ElectricalBranch
  nodes
    p = foundation.electrical.electrical; % +:left
    n = foundation.electrical.electrical; % +:right
  end
  variables
    i = { 0, 'A' };
    v = { 0, 'V' };
  end
  branches
    i : p.i -> n.i;
  end
  equations
    v == p.v - n.v;
  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
  equations
      assert(c>0, 'Capacitance must be greater than zero');
      i == c * v.der;
  end
end
```

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

Importing Domain and Component Classes

You must store Simscape model files (domains and components) in package directories, as described in “Organizing Your Simscape Files” on page 4-25. Like the MATLAB class system, each package defines a scope (or namespace). You can uniquely identify a model class name and access it using a fully qualified reference. For example, you can access the domain model class `electrical` using `foundation.electrical.electrical`.

In composite components, class member declarations include user-defined types, that is, component classes. If you do not use `import` statements, accessing component class names from a different scope always requires a fully qualified reference. For example, the Foundation library Resistor block is:

```
foundation.electrical.elements.resistor
```

An import mechanism provides a convenient means to accessing classes defined in different scopes, with the following benefits:

- Allows access to model class names defined in other scopes without a fully qualified reference
- Provides a simple and explicit view of dependencies on other packages

There are two types of syntax for the `import` statement. One is a qualified import, which imports a specific package or class:

```
import package_or_class;
```

The other one is an unqualified import, which imports all subpackages and classes under the specified package:

```
import package.*;
```

The package or class name must be a full path name starting from the library root (the top-level package directory name) and containing subpackage names as necessary.

You must place `import` statements at the beginning of a Simscape file. The scope of imported names is the entire Simscape file, except the `setup` section. For example, if you use the following `import` statement:

```
import foundation.electrical.elements.*;
```

at the beginning of your component file, you can refer to the Foundation library Resistor block elsewhere in this component file directly by name:

```
rotorResistor = resistor(R = rotor_resistance);
```

See the `import on page 5-44` reference page for syntax specifics. For an example of using `import` statements in a custom component, see the “Transmission Line” example. To view the Simscape file, open the example, then double-click **Open the transmission line component library**. In the TransmissionLine_lib window, double-click the T-Section Transmission Line block and then, in the block dialog box, click **Source code**.

See Also

Related Examples

- “Composite Component Using import Statements” on page 2-110

Composite Component Using import Statements

This example shows how you can use `import` statements to implement a composite component equivalent to the one described in “Composite Component — DC Motor” on page 2-95 . The two components are identical, but, because of the use of the `import` statements, the amount of typing in the `nodes` and `components` sections is significantly reduced.

```
import foundation.electrical.electrical;   % electrical domain class definition
import foundation.electrical.elements.*;   % electrical elements
import foundation.mechanical.rotational.*; % mechanical rotational domain and elements
component DC_Motor1
% DC Motor1
% 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 = electrical;               % +:left
    n = electrical;               % -:left
    R = rotational;               % R:right
    C = rotational;               % C:right
end

parameters
    rotor_resistance    = { 3.9, 'Ohm' };           % Rotor Resistance
    rotor_inductance    = { 12e-6, 'H' };           % Rotor Inductance
    motor_inertia       = { 0.01, 'g*cm^2' };       % Inertia
    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
    breakaway_velocity  = { 0.1, 'rad/s' };         % Breakaway friction velocity
    back_emf_constant   = { 0.072e-3, 'V/rpm' };    % Back EMF constant
end

components(ExternalAccess=observe)
    rotorResistor                 = resistor(R = rotor_resistance);
    rotorInductor                 = inductor(l = rotor_inductance);
    rotationalElectroMechConverter = rotational_converter(K = back_emf_constant);
    friction                      = friction(brkwy_trq = breakaway_torque, Col_trq = coulomb_torque, ...
                                         visc_coef = viscous_coeff, brkwy_vel = breakaway_velocity);
    motorInertia                  = inertia(inertia = motor_inertia);
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

end
```

Consider the three `import` statements at the beginning of the file. The first one:

```
import foundation.electrical.electrical;
```

is a qualified import of the Foundation electrical domain class. Therefore, in the `nodes` section, you can define the `p` and `n` nodes simply as `electrical`.

The second statement:

```
import foundation.electrical.elements.*;
```

is an unqualified import, which imports all subpackages and classes under the `foundation.electrical.elements` subpackage and therefore gives you direct access to all the Foundation electrical components in the Elements sublibrary, such as `inductor`, `resistor`, and `rotational_converter`.

The third statement:

```
import foundation.mechanical.rotational.*;
```

is an unqualified import, which imports all subpackages and classes under the `foundation.mechanical.rotational` subpackage and therefore gives you direct access to the Foundation mechanical rotational domain definition (`rotational`) and components (such as `friction` and `inertia`).

The `nodes` block declares two electrical nodes, `p` and `n`, and two mechanical rotational nodes, `R` and `C`.

The `components` block declares all the member (constituent) components, using the following components from the Simscape Foundation library:

- Resistor
- Inductor
- Rotational Electromechanical Converter
- Rotational Friction
- Inertia

Because of the `import` statements at the top of the file, these classes already exist in the scope of the file, and you do not have to specify their complete names starting from the top-level package directory.

See Also

Related Examples

- “Composite Component — DC Motor” on page 2-95

More About

- “Importing Domain and Component Classes” on page 2-108

Advanced Techniques

Mode Chart Modeling

In this section...
“About Mode Charts” on page 3-2
“Mode Chart Syntax” on page 3-2
“Mode Chart Example” on page 3-3

About Mode Charts

Mode charts provide an intuitive way to model components characterized by a discrete set of distinct operating modes. A car clutch is a good example of such a component. It has several operating modes, with each mode being defined by a different set of equations. It also has a transition logic, with a set of predicate conditions defining when the clutch transitions from one mode to another. It is possible to model this component using primitive constructs, such as event variables and `edge` operators, but this way of modeling lacks readability. For more complex components, the file becomes cumbersome and unwieldy. Every time you model a component with multiple operating modes and transitions, this component is a good candidate for a mode chart implementation.

These constructs in Simscape language let you perform mode chart modeling:

- `modecharts` — A top-level section in a component file. It can contain one or more `modechart` constructs.
- `modechart` — A named construct that contains a textual representation of the mode chart: modes, transitions, and an optional initial mode specification.
- `modes` — A section in a mode chart that describes all the operating modes. It can contain one or more `mode` constructs.
- `mode` — A named construct that corresponds to a distinct operating mode of the component, defined by a set of equations.
- `transitions` — A section in a mode chart that describes transitions between the operating modes, based on predicate conditions.
- `initial` — An optional section in a mode chart that specifies the initial operating mode, based on a predicate condition. If the predicate is not true, or if the `initial` section is missing, then the first mode listed in the `modes` section is active at the start of simulation.
- `entry` — An optional section inside a `mode` construct in a mode chart that lets you specify the actions to be performed upon entering the mode.

Mode Chart Syntax

In its simplest form, the hierarchical structure of a `modecharts` section can look like this:

```
modecharts
   mc1 = modechart
      modes
         mode m1
            equations
               ...
            end
         end
         mode m2
```

```
               equations
                  ...
               end
            end
         end
         transitions
            m1->m2 : p1;
         end
         initial
            m2 : p2;
         end
      end
   end
```

It contains one mode chart, `mc1`, with two modes, `m1` and `m2`.

The system transitions from mode `m1` to mode `m2` when the predicate condition `p1` is true.

If the predicate condition `p2` is true, the simulation starts in mode `m2`, otherwise in mode `m1`.

In this example, the `transitions` section does not define a transition from mode `m2` to mode `m1`. Therefore, according to this mode chart, once the system reaches mode `m2`, it never goes back to mode `m1`.

Mode Chart Example

Use this simple example to understand how the mode charts work. For a more detailed example, see “Switch with Hysteresis” on page 3-5.

```
component ExampleChart

  inputs
     u1 = 0;
  end

  outputs
     y = 0;
  end

  parameters
     p = 1;
  end

  modecharts(ExternalAccess = observe)
     mc1 = modechart
        modes
           mode m1
              equations
                 y==1;
              end
           end
           mode m2
              equations
                 y==2;
              end
              end
           mode m3
```

```
              equations
                 y==3;
              end
           end
        end
        transitions
           m1->m2 : u1<0;
           m2->m3 : u1>0;
        end
        initial
           m2 : p<0;
        end
     end
  end

end
```

The component implements a simple chart with three operating modes:

- In the first mode, the output signal equals 1.
- In the second mode, the output signal equals 2.
- In the third mode, the output signal equals 3.

The component transitions from the first to the second mode when the input signal is negative, and from the second to the third mode when the input signal is positive.

The initial mode depends on the block parameter value: if parameter `p` is negative, simulation starts with the block in the second mode, otherwise — in the first mode.

See Also

`entry` | `initial` | `modecharts` | `modes` | `transitions`

More About

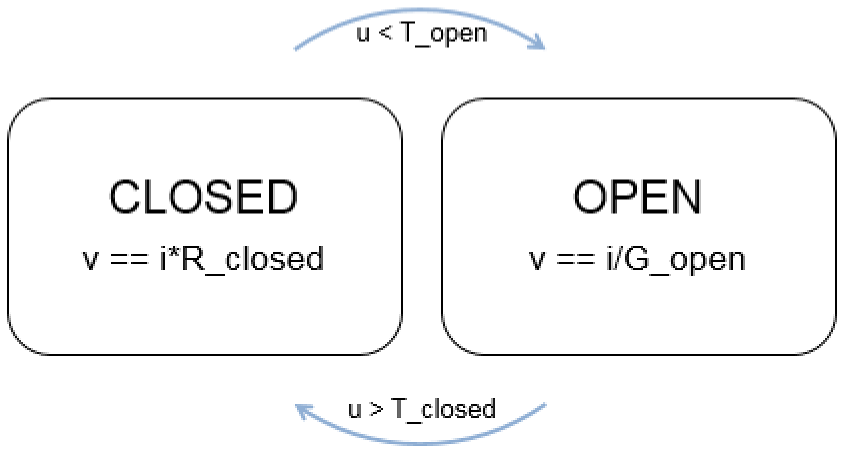
- “Switch with Hysteresis” on page 3-5
- “State Reset Modeling” on page 3-11

Switch with Hysteresis

The Switch block in the Simscape Foundation library implements a switch controlled by an external physical signal. The block uses an `if-else` statement. If the external physical signal at the control port is greater than the threshold, then the switch is closed, otherwise the switch is open.

This example implements a switch with hysteresis applied to the switching threshold level. The hysteresis acts to prevent rapid spurious switching when the control signal is noisy.

The switch has two distinct operating modes, shown in the diagram. If the external physical signal at the control port is greater than the upper threshold, then the switch is closed. If the signal is lower than the lower threshold, the switch is open.



The following component implements the logic in the diagram by using a mode chart.

```
component delayed_switch
% Switch with Hysteresis

inputs
    u = { 0.0, '1' };
end

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

parameters
    R_closed = { 0.01, 'Ohm' };   % Closed resistance R_closed
    G_open   = { 1e-8, '1/Ohm' }; % Open conductance G_open
    T_closed = { 0.5, '1' };      % Upper threshold
    T_open   = { 0, '1' };        % Lower threshold
    InitMode = switching.open;    % Initial Mode
end

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

branches
    i : p.i -> n.i;
end

% Validate parameter values
equations
            assert( T_closed >= T_open, 'Upper threshold must be higher than Lower threshold' );
end
```

```
modecharts(ExternalAccess = observe)
    m1 = modechart
        modes
            mode CLOSED
                equations
                    v == p.v - n.v;
                    v == i*R_closed;
                end
            end
            mode OPEN
                equations
                    v == p.v - n.v;
                    v == i/G_open;
                end
            end
        end
        transitions
            CLOSED -> OPEN : u < T_open;
            OPEN -> CLOSED : u > T_closed;
        end
        initial
            OPEN : InitMode <= 0;
        end
    end

end

end
```

The mode chart `m1` defines two modes, `CLOSED` and `OPEN`. Each mode has an `equations` section that lists all the applicable equations. The `transitions` section defines the transitions between the operating modes, based on predicate conditions:

- The switch transitions from `CLOSED` to `OPEN` when the control signal falls below the lower threshold, `T_open`.
- The switch transitions from `OPEN` to `CLOSED` when the control signal rises above the upper threshold, `T_closed`.

The `initial` section specifies the initial operating mode, based on a predicate condition:

- If the predicate is true (that is, the **Initial Mode** parameter value is less than or equal to 0), then the `OPEN` mode is active at the start of simulation.
- If the predicate is not true, then the `CLOSED` mode (the first mode listed in the `modes` section) is active at the start of simulation.

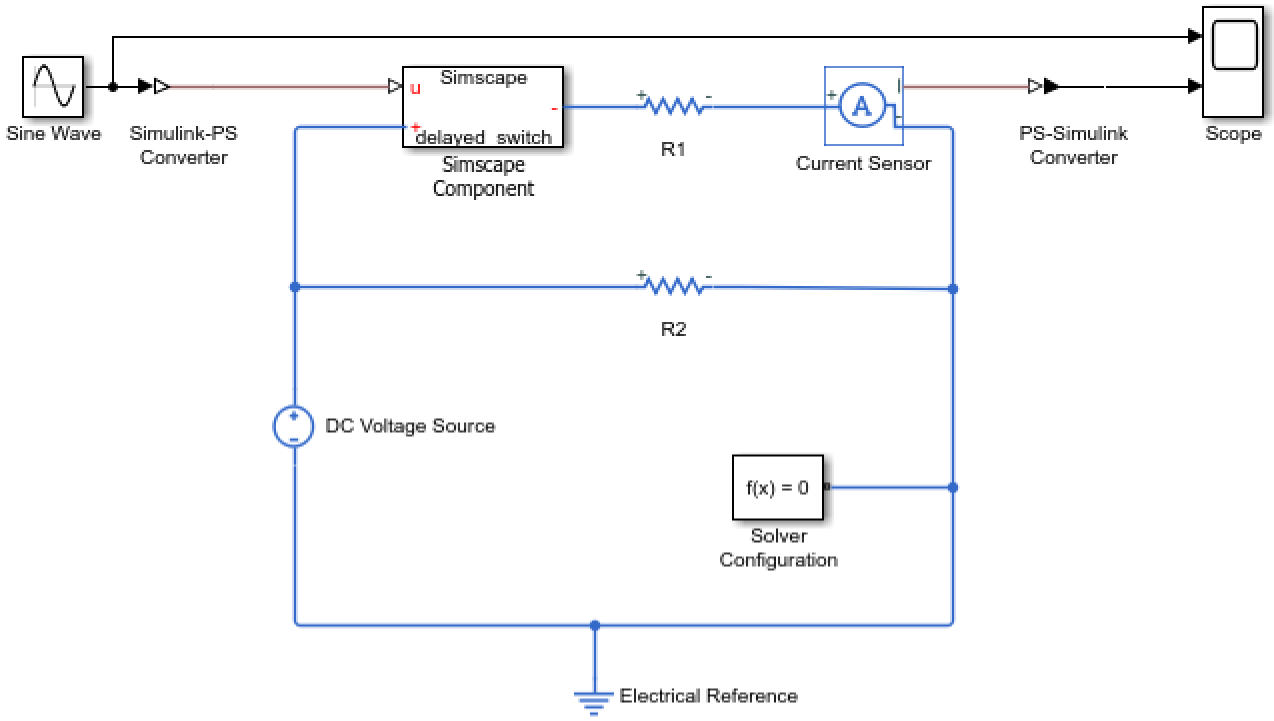
Note The **Initial Mode** parameter uses an enumeration:

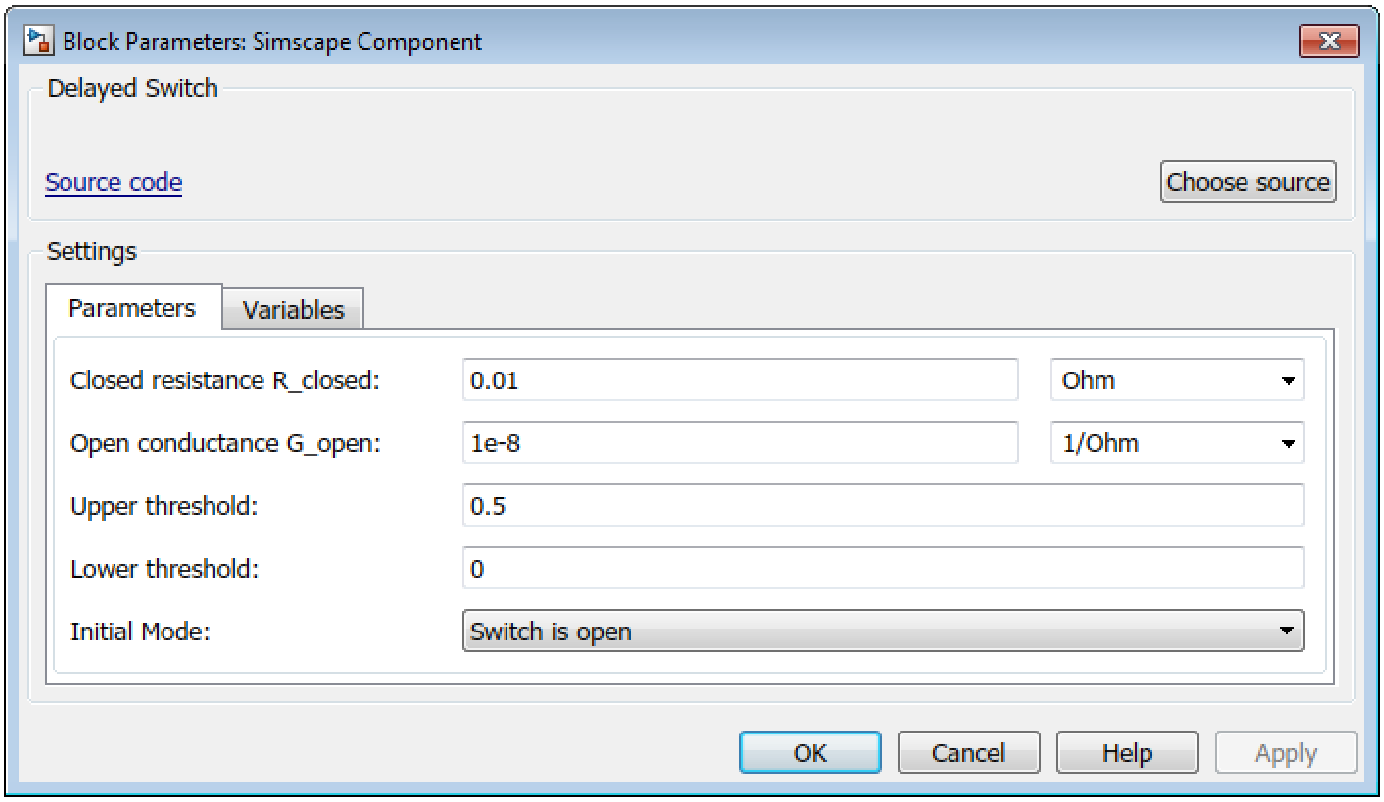
```
classdef switching < int32
   enumeration
     open (0)
     closed (1)
   end
   methods(Static)
       function map = displayText()
         map = containers.Map;
         map('open') = 'Switch is open';
         map('closed') = 'Switch is closed';
       end
   end
 end
```

For the component to work as described, this enumeration needs to be in a separate `switching.m` file. The file can be located either on the MATLAB path or in a package imported into the component.

In general, enumerations are very useful in mode charts, because they let you specify a discrete set of acceptable parameter values. For more information, see “Enumerations” on page 3-15.

To verify the correct component behavior, deploy it in a Simscape Component block. Create a simple test model, as shown, with all the blocks using the default parameter values.




Block Parameters: Simscape Component

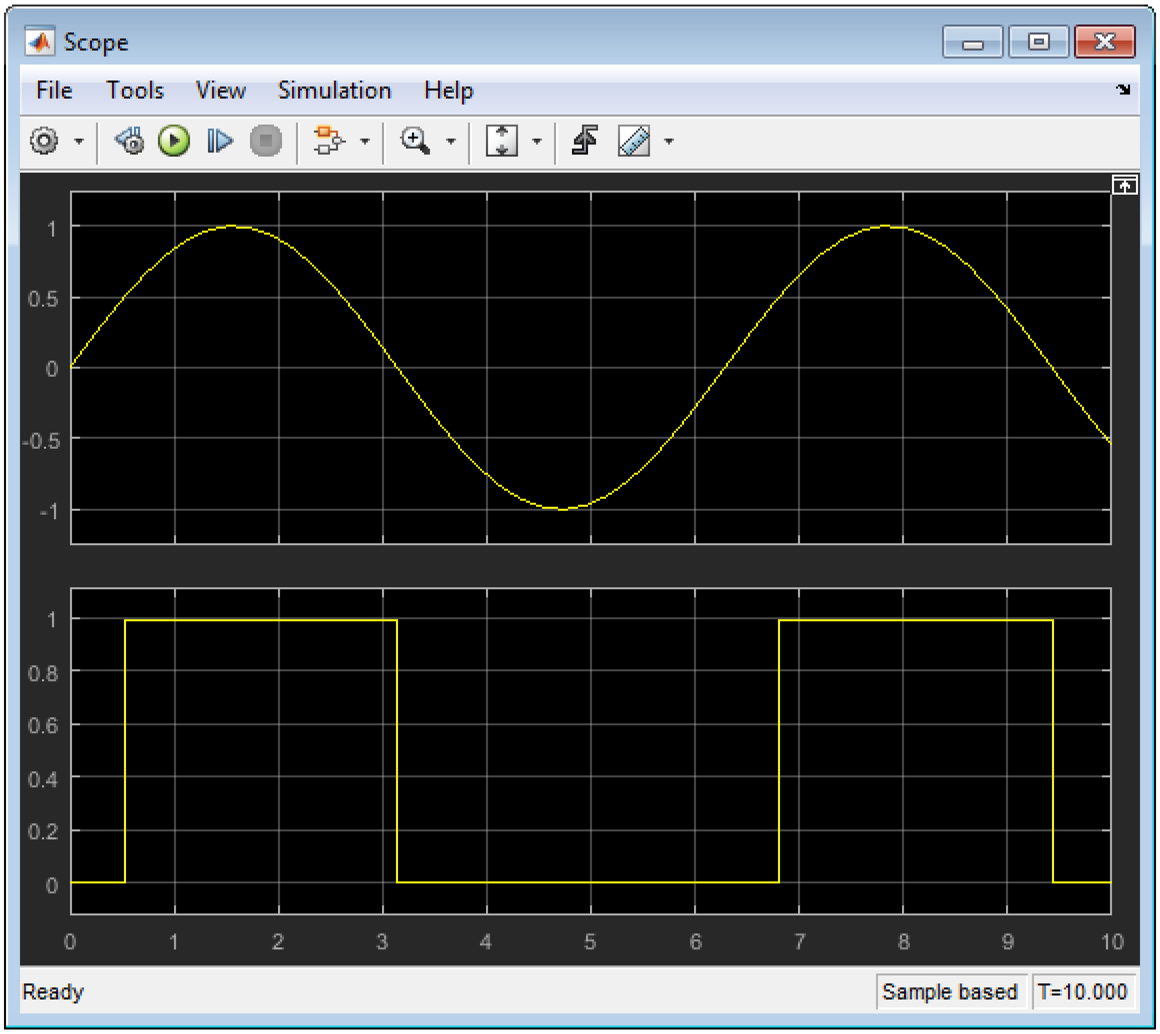
Delayed Switch

Source code

Settings

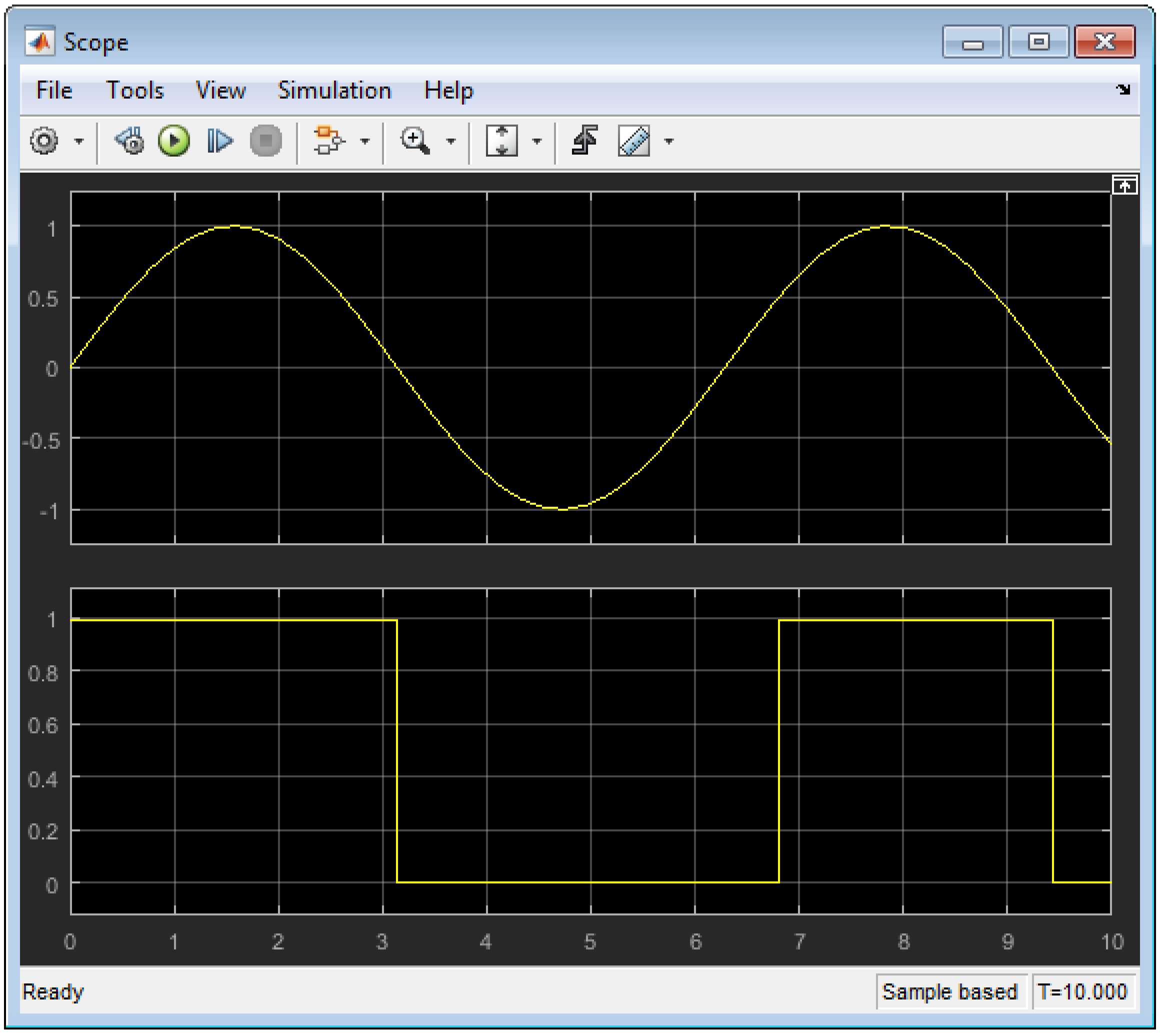
Closed resistance R_closed:	0.01	Ohm
Open conductance G_open:	1e-8	1/Ohm
Upper threshold:	0.5	
Lower threshold:	0	
Initial Mode:	Switch is open	

Simulate the model with the default values.



The **Initial Mode** parameter value is `Switch is open`. This enumerated value evaluates to 0, which makes the predicate in the `initial` section true. Therefore, at the start of simulation the switch is open and no current flows through the resistor R1. When the control signal value reaches 0.5 (the **Upper threshold** parameter value), the switch closes and the current through the branch, based on the other parameter values, is 1A. When the control signal falls below 0 (the **Lower threshold** parameter value), the switch opens.

Now change the **Initial Mode** parameter value to `Switch is closed` and simulate the model. The enumerated value evaluates to 1, the predicate condition in the `initial` section is no longer true, and therefore the first mode listed in the `modes` section is active. At the start of simulation, the switch is closed, and it stays closed until the control signal falls below 0.



See Also

`initial` | `modecharts` | `modes` | `transitions`

More About

- "Mode Chart Modeling" on page 3-2
- "Enumerations" on page 3-15

State Reset Modeling

In this section...
"About State Reset" on page 3-11
"State Reset Example" on page 3-11

About State Reset

Event-based methods of state reinitialization and impulse handling let you model physical phenomena such as collisions and bouncing balls. Using state reset methods provides a significant boost in simulation speed for such models, compared to continuous simulation.

To implement a state reset, mode charts can contain instantaneous modes and compound transitions. An instantaneous mode is a mode that is active only for one event iteration. You specify that a mode is instantaneous by using a compound transition:

```
A -> B -> C : t
```

The middle mode, `B`, is instantaneous. When predicate `t` becomes true, the system transitions from mode `A` to mode `B`, performs one event iteration, and then immediately transitions to mode `C`.

You declare instantaneous modes the same way as regular modes, using the `mode` section of a mode chart. To specify that a mode is instantaneous, list it as the middle mode in a compound transition. Only one instantaneous mode is allowed per transition, therefore, a compound transition cannot contain more than three modes.

In the majority of state reset use cases, the reset value is a function of the previous value of the variable. For example, when modeling a bouncing ball, the new velocity depends on the velocity before impact. The `entry` section, declared within a `mode` section in a mode chart, lets you specify the actions to be performed upon entering the mode. These actions are event variable updates based on the value of a continuous expression immediately before entering the mode. When modeling state reset, you can use entry actions to update the value of an event variable based on the value of the respective continuous variable immediately before entering the mode.

When you connect multiple ideal components that use state reset, the solver automatically detects and propagates impulses in continuous states during variable reinitialization. Impulse propagation can only trigger events whose predicates are linear expressions of continuous states. Also, impulse detection can add computational cost during transient initialization. Two options in the Solver Configuration block, **Compute impulses** and **Impulse iterations**, let you control the computational cost of impulse detection during transient initialization. If you use fixed-cost simulation for a model that contains components with state reset, select the **Compute impulses** check box to get the correct impulse propagation results.

State Reset Example

Use this simple example to understand how to model state reset. For a more detailed example, see "Mass on Cart using an Ideal Hard Stop".

The Translational Hard Stop block in the Simscape Foundation library models a hard stop as a spring and damper that come into contact with the slider at the bounds.

This example implements an ideal translational hard stop, where the slider velocity resets instantaneously upon hitting the upper or lower bound.

```
component ideal_hard_stop

nodes
    R = foundation.mechanical.translational.translational % R:left
    C = foundation.mechanical.translational.translational % C:right
end

parameters
    upper_bnd = { 0.1, 'm'} % Upper bound
    lower_bnd = {-0.1, 'm'} % Lower bound
    e = 0.8                 % Coefficient of restitution
end

variables
    v = {0, 'm/s'}   % Velocity
    f = {0, 'N'}     % Force
    x = {value = {0, 'm'}, priority = priority.high} % Position
end

variables(Event = true, Access = private, ExternalAccess = none)
    v_old = {0, 'm/s'}
end

branches
    f : R.f -> C.f
end

equations
    v == R.v - C.v
    x.der == v

    assert(e > 0);
    assert(e <= 1);
    assert(upper_bnd > 0);
    assert(lower_bnd < 0);
end

modecharts(ExternalAccess = observe)
  m = modechart
    modes
      mode FREE
        equations
          f == 0
        end
      end
      mode IMPACT
        entry
          v_old = v
        end
        equations
          v == -e*v_old
        end
      end
    end
    transitions
```

```
      FREE -> IMPACT -> FREE : x <= lower_bnd && v < 0
      FREE -> IMPACT -> FREE : x >= upper_bnd && v > 0
    end
  end
end

end
```

The mode chart `m` defines two modes:

- `FREE`, when the slider travels freely between the bounds.
- `IMPACT`, when the slider hits one of the bounds.

Each mode has an `equations` section that lists the equations applicable to that mode. The `equations` section outside the mode chart lists the asserts and equations that apply to both modes.

The `transitions` section defines two compound transitions, one for the slider hitting the lower bound and one for the upper bound. In each transition, when the predicate becomes true, the component switches from the `FREE` mode to `IMPACT`, and then back to `FREE`. `IMPACT` is an instantaneous mode.

When the component enters the `IMPACT` mode, the event variable `v_old` gets updated with the value of velocity before impact. This update action is defined in the `entry` section for that mode. Then, in the `equations` section for this mode, the velocity, `v`, is reset to a value that is a function of this previous velocity value and the coefficient of restitution, `e`.

The component implements separate transitions for the upper and lower bounds to improve code readability. The predicate for each of these compound transitions includes both the slider position and the velocity sign, to avoid entering a self-loop. Compound transitions follow the same rules as regular transitions. If a predicate is true, the system immediately enters the transition. Therefore, if you defined a compound transition based only on the slider position:

```
transitions
    FREE -> IMPACT -> FREE : x <= lower_bnd || x >= upper_bnd
end
```

the predicate could still be true after completing the transition, the system would enter an infinite loop and eventually generate an error. To avoid this situation, it is a good practice to try to model compound transitions in such a way that the instantaneous mode invalidates the predicate:

```
transitions
    FREE -> IMPACT -> FREE : x <= lower_bnd && v < 0
    FREE -> IMPACT -> FREE : x >= upper_bnd && v > 0
end
```

In this case, while in the instantaneous mode, the velocity flips sign and the predicate is no longer valid.

The “Mass on Cart using an Ideal Hard Stop” example uses a custom Ideal Hard Stop block with additional options that cover a wider variety of use cases. This block has a more complex mode chart, but the modeling principles and the block behavior are similar.

See Also

`entry` | `initial` | `modecharts` | `modes` | `transitions`

More About

- “Mode Chart Modeling” on page 3-2

Enumerations

In this section...

Enumerations in Simscape Language

Simscape language supports MATLAB enumerations in:

- Component parameters
- Event variables and `when` clauses
- Equation predicates
- Conditional declaration predicates
- Function arguments (such as an interpolation method in `tablelookup`)
- Mode charts

You define enumerations using a MATLAB enumeration class. For more information, see "Enumerations".

The enumeration class must derive from the `int32` type, for example:

```
classdef offon < int32
   enumeration
     off (0)
     on (1)
   end
end
```

Save the enumeration class definition in a .m file with the same name as the class. For more information, see "Rules and Restrictions" on page 3-20.

You can then use this enumeration in a component parameter:

```
parameters
   fl_c = offon.off; % Fluid compressibility
end
```

In the resulting block dialog, the **Fluid compressibility** parameter will have a drop-down list of values, `off` and `on`, with `off` as the default.

Specifying Display Strings for Enumeration Members

When using enumerations in component parameters, you can specify user-friendly strings to be displayed in the block dialog, instead of member identifiers:

```
classdef damping < int32
   enumeration
     direct (0)
     derived (1)
   end
   methods(Static)
       function map = displayText()
         map = containers.Map;
         map('direct') = 'By damping value';
         map('derived') = 'By no-load current';
       end
   end
end
```

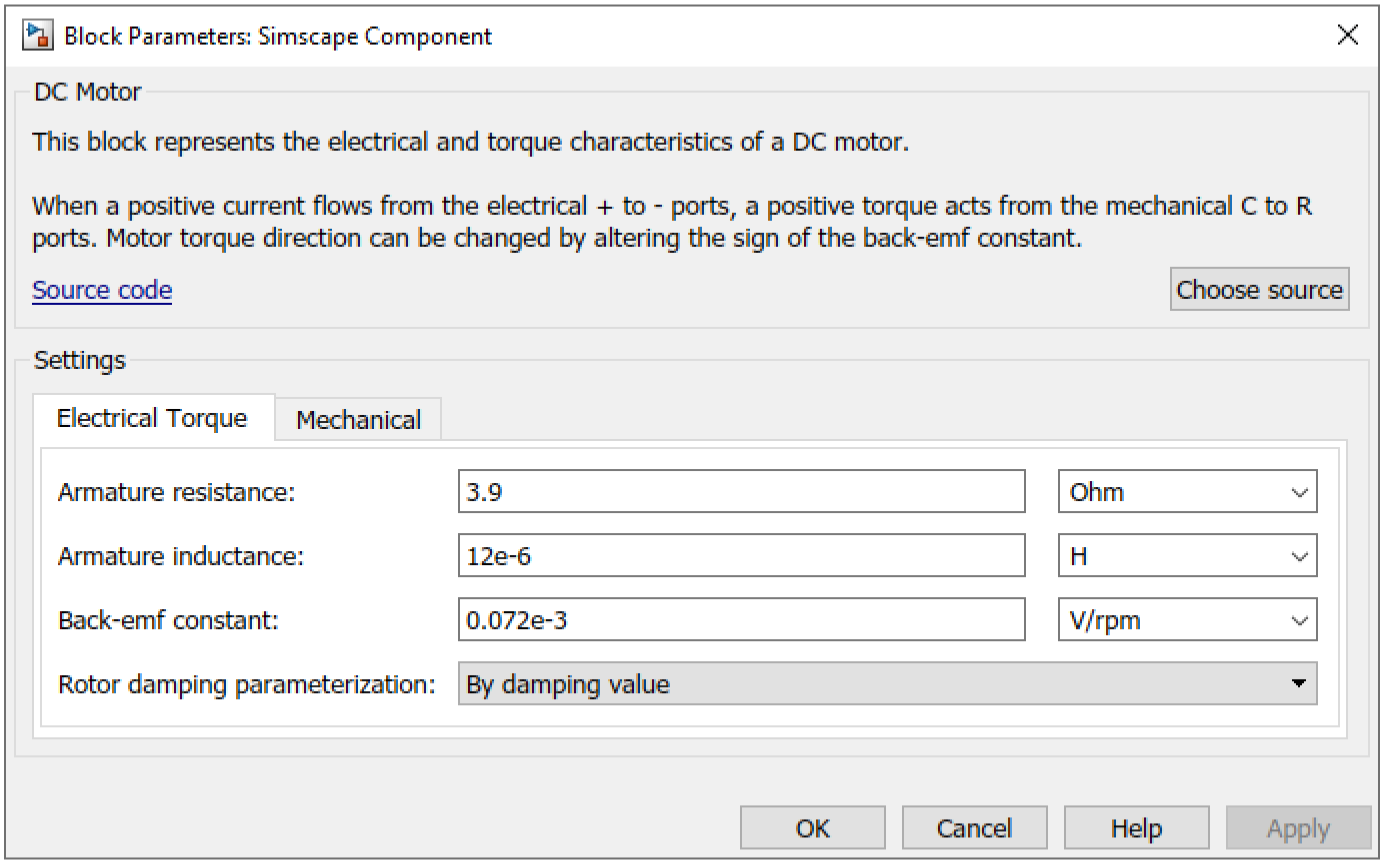
You can then use this enumeration in a component parameter, for example:

```
parameters
   r_damp = damping.direct; % Rotor damping parameterization
end
```

In the resulting block dialog, the **Rotor damping parameterization** parameter has a drop-down list of values:

- `By damping value`
- `By no-load current`

`By damping value` is the default value.


Block Parameters: Simscape Component

DC Motor

This block represents the electrical and torque characteristics of a DC motor.

When a positive current flows from the electrical + to - ports, a positive torque acts from the mechanical C to R ports. Motor torque direction can be changed by altering the sign of the back-emf constant.

Source code

Settings

Armature resistance:	3.9	Ohm
Armature inductance:	12e-6	H
Back-emf constant:	0.072e-3	V/rpm
Rotor damping parameterization:	By damping value	

For a detailed example of using enumeration with display strings in a component parameter, see "Use Advanced Techniques to Customize Block Display" on page 4-47.

Evaluating Enumeration Members

If an enumeration class derives from a built-in numeric class, the subclass inherits ordering and arithmetic operations that you can apply to the enumerated names. Enumeration classes used in Simscape language must derive from the `int32` type. Therefore, when used in mathematical expressions, enumeration members convert to integers according to the specified value. For example, the "Switch with Hysteresis" on page 3-5 component uses this enumeration:

```
classdef switching < int32
   enumeration
     open (0)
     closed (1)
   end
   methods(Static)
       function map = displayText()
         map = containers.Map;
         map('open') = 'Switch is open';
         map('closed') = 'Switch is closed';
       end
   end
 end
```

The enumeration is used in the **Initial Mode** parameter declaration:

```
parameters
    ...
    InitMode = switching.open;    % Initial Mode
end
```

Then, the `initial` section of the mode chart uses the **Initial Mode** parameter value in the predicate expression:

```
initial
   OPEN : InitMode <= 0;
end
```

When the **Initial Mode** parameter value is `Switch is open`, the corresponding enumeration member, `open (0)`, evaluates to 0, and the predicate is true. Therefore, at the start of simulation the switch is open.

Conversely, when the parameter value is `Switch is closed`, the corresponding enumeration member, `closed (1)`, evaluates to 1, and the predicate is false. For more information, see “Switch with Hysteresis” on page 3-5.

Using Enumeration in Event Variables and when Clauses

The previous sections discussed using enumerations to declare component parameters with a discrete set of acceptable values. However, you can also use enumerations to declare event variables, because they also have a discrete set of values.

Event variables are piecewise constant, that is, they change values only at event instants (by using the `when` clause), and keep their values constant between events.

For example:

```
variables (Event = true)
   x = myEnum.a;
end
events
   when edge(time > {1.0, 's'})
     x = myEnum.b;
   end
end
```

Using Enumeration in Predicates

The “Switch with Hysteresis” on page 3-5 component shows an example of using an enumerated parameter in a mode chart predicate.

Another good practice is using enumerated parameters in conditional declaration predicates, to define block variants. For example, you can have two variants of a pipe, one that accounts for resistive properties only and the second that also models fluid compressibility:

```
component MyPipe
  parameters
     fl_c = offon.off; % Fluid compressibility
  end
  [...] % other parameters, variables, branches
  if fl_c == offon.off
    equations
       % first set of equations, resistive properties only
    end
  else
    variables
       % additional variable declarations, needed to account for fluid compressibility
    end
    equations
       % second set of equations, including fluid compressibility
    end
  end
end
```

In this example, the block parameter **Fluid compressibility** is using the `offon` enumeration:

```
classdef offon < int32
   enumeration
     off (0)
     on (1)
   end
end
```

In the resulting block dialog, the **Fluid compressibility** parameter has a drop-down list of values, `off` and `on`, with `off` as the default. If the parameter is set to `off`, the first set of equations gets activated and the block models only the resistive properties of the pipe. If the block user changes the value of the parameter, then the `else` branch gets activated, and the compiled model includes the additional variables and equations that account for fluid compressibility. For more information on defining block variants, see “Defining Component Variants” on page 2-75.

Likewise, you can use enumerated parameters and event variables in equation predicates:

```
parameters
   p = myEnum.a;
end
variables
   x = 0;
   y = 0;
end
equations
   if p == myEnum.a
     y == x * 100;
   elseif p == myEnum.b
     y == x * 0.01;
   else    % (p == myEnum.c)
     y == x;
   end
end
```

Using Enumeration in Function Arguments

Another way to use enumerations is in function arguments. For example, the `tablelookup` function has two interpolation methods, `linear` and `smooth`, and three extrapolation methods, `linear`, `nearest`, and `error`.

The Foundation library includes built-in enumerations, `interpolation.m` and `extrapolation.m`:

```
classdef interpolation < int32
   enumeration
       linear (1)
       smooth (2)
   end
   methods(Static)
    function map = displayText()
      map = containers.Map;
      map('linear') = 'Linear';
      map('smooth') = 'Smooth';
    end
  end
end

classdef extrapolation < int32
   enumeration
       linear (1)
       nearest (2)
       error (3)
   end
   methods(Static)
    function map = displayText()
      map = containers.Map;
      map('linear') = 'Linear';
      map('nearest') = 'Nearest';
      map('error') = 'Error';
    end
  end
end
```

These enumerations are located in the directory *`matlabroot`*`\toolbox\physmod\simscape\library\m\+simscape\+enum`.

You can use these enumerations to declare component parameters, and then use these parameters as function arguments:

```
parameters
    interp = simscape.enum.interpolation.linear; % Interpolation method
    extrap = simscape.enum.extrapolation.linear; % Extrapolation method
end
equations
    o == tablelookup(xd, yd, x, interpolation = interp_method, extrapolation = extrap_method);
end
```

Instead of providing fully qualified names, you can use the `import` statement to reduce the amount of typing:

```
import simscape.enum.*
...
parameters
    interp = interpolation.linear; % Interpolation method
    extrap = extrapolation.linear; % Extrapolation method
end
equations
    o == tablelookup(xd, yd, x, interpolation = interp, extrapolation = extrap);
end
```

Rules and Restrictions

Enumeration definitions are global. You define an enumeration once, in a separate file, and can then use the same enumeration in multiple components.

The file containing the enumeration class definition must reside on the MATLAB path or in a package directory. For more information about package directories, see “Organizing Your Simscape Files” on page 4-25.

Parameters that have enumerated values are marked as `Compile-time` only in the block dialogs.

Similar to MATLAB enumerations, you can define more than one identifier for the same integer value, for example:

```
classdef myColor < int32
   enumeration
     red (0)
     blue (1)
     yellow (2)
     green (0)
   end
 end
```

The first identifier in the enumeration block with a given integer value is the actual identifier, and subsequent identifiers are aliases.

Note Although multiple identifiers with the same integer value are allowed, it is recommended that you use unique integer values within a Simscape language enumeration set, for better clarity.

See Also

Related Examples

- “Switch with Hysteresis” on page 3-5

Declaration Functions

In this section...

You can use declaration functions to compute derived parameter values or initialize variables, instead of doing this inside the `setup` function.

Note Starting in R2019a, using `setup` is not recommended. Other constructs available in Simscape language let you achieve the same results without compromising run-time capabilities. For more information, see “setup is not recommended” on page 5-69.

Declaration function is a MATLAB function used inside a member declaration section in a Simscape file. A declaration function can be any MATLAB function (even if it is not supported in the Simscape language `equations` section), including user-defined functions on the MATLAB path. For example:

```
component A
  parameters
    p1 = 1;
    p2 = 0;
  end
  parameters(Access = private)
    pDerived = gamma(p1) + p2;
  end
  variables(Access = private)
    vDerived = {value = {my_fcn(p1,p2) + 1, 'm'}, priority = priority.high };
  end
  equations
    ...
  end
end
```

Use the `Access=private` attribute for member declaration unless all the arguments of the declaration function are constants.

Exercise caution when using persistent variables inside a declaration function, because this may lead to inconsistent results for multiple simulation runs.

Multiple Return Values

Declaration functions can return multiple values. They follow the general MATLAB function conventions for multiple return values. For example, if `my_fcn()` is a declaration function that returns three values:

```
[id1, ~, id3] = my_fcn();  % omit the second return value
```

```
[id1] = my_fcn();  % rules of single assignment apply, nonrequested return values ignored
```

The following restrictions apply:

- You can use multiple value assignments on the left-hand side only for parameters and variables with the `Access=private` attribute.
- When omitting return values using the placeholder attribute (~), at least one value must be assigned. Empty declarations produce an error in Simscape language.

Restriction on Values with Units

Inputs and outputs of a declaration function must be unitless, that is, have a unit of `'1'`. Therefore, you cannot directly pass parameter values, with units, as declaration function inputs.

For example, parameter `p` has the units of `'m'`. To use it as an input for the `myfcn` function, use the `value` function to get the unitless value of the parameter.

```
parameters
      p = {1,'m'}
end
parameters(Access = private)
      pd = my_fcn(value(p,'m'));   % extract unitless value from p
end
```

In the previous example, `pd` is a unitless parameter. To declare it as a value with unit, use the `{value,'unit'}` syntax, for example:

```
      pd = {my_fcn(value(p,'m')),'m/s'};
```

For multiple input and return values with units, use this syntax:

```
      [y_value,z_value] = my_fcn(value(a,'V'),value(b,'V'));
      y = {y_value,'V'};
      z = {z_value,'V'};
```

For more information, see “Declaring a Member as a Value with Unit” on page 2-5.

Run-Time Compatibility

Member declarations for parameters and variables can include calls to MATLAB functions that generate code.

By default, the declaration function will be evaluated at run time if a run-time parameter appears in its input parameters. Otherwise, it will be evaluated at compile time.

In this example, `my_fcn` is a MATLAB function that supports code generation:

```
component A
  parameters
    p1 = 1;
    p2 = 0;
  end
  parameters(Access = private)
    pDerived = my_fcn(p1,p2);
  end
  equations
    ...
  end
end
```

If `p1` or `p2` is designated as `Run-time` in the block dialog, then `my_fcn` is evaluated at run time, and you can tune these parameter values without regenerating code.

If `my_fcn` does not support code generation, you can set the member attribute`MATLABEvaluation=compiletime`, to prevent the block user from accidentally designating any of the function input parameters as `Run-time` in the block dialog:

```
component A
  parameters
    p1 = 1;
    p2 = 0;
  end
  parameters(Access = private,MATLABEvaluation = compiletime)
    pDerived = my_fcn(p1,p2);
  end
  equations
    ...
  end
end
```

If you set this attribute, the declaration function will be evaluated only at compile time, and the block parameters `p1` and `p2` will be marked as `Compile-time` only.

To work with run-time parameters:

- The declaration function must be in an unprotected MATLAB file
- All MATLAB code called must be MATLAB Coder™ compatible
- Subfunctions can be in protected MATLAB files, but to use them with run-time parameters:
  - Use `coder.allowpcode('plain')`
  - Turn on `lint`: `%#codegen`

For more information, see “Run-Time Parameters”.

Simscape Functions

In this section...

Simscape functions model a class of pure first-order mathematical functions with explicit input-output relationship. These functions explicitly map the inputs of numerical values into outputs of numerical values by using declarative expressions. When a component calls a Simscape function, numerical input values are passed to the function, which then evaluates these declarative expressions to compute the output values.

There are two types of Simscape functions:

- Main function — In general, the purpose of Simscape functions is to reuse expressions in equations of multiple components, as well as in member declarations of domain or component files. Each of these functions must be in a separate Simscape file, with the file name matching the function name.
- Local function — In contrast, local Simscape functions reside inside a Simscape file that defines a component, domain, or another function, and are accessible only by that component, domain, or main function. For more information, see "Local Simscape Functions" on page 3-27.

Main Simscape Functions

Each main function must be in a separate Simscape file. The file name must match the function name. For example, function `foo` must be in a file called `foo.ssc`.

The Simscape function file must start with the keyword `function`, followed by the function header, which includes the function name, inputs, and outputs. For example:

```
function out = MyFunction(in1,in2)
```

If the function has multiple return values, the syntax is:

```
function [out1,out2] = MyFunction(in1,in2)
```

The body of the function must be enclosed inside the `definitions` section, for example:

```
function out = SumSquared(in1,in2)
   definitions
      out = in1^2 + 2*in1*in2 + in2^2;
   end
end
```

Syntax Rules

- One or more output parameters are allowed.
- If an output parameter is not used on the left-hand side of the `definitions` section, you get an error.

- Zero or more input parameters are allowed.
- When the function is called, the number of input arguments must match the number of input parameters.
- Input parameters are positional. This means that the first input argument during the function call is passed to the first input parameter, and so on. For example, if you write an equation:

  ```
  o == SumSquared(5,2);
  ```

  then `in1` is 5 and `in2` is 2.
- If the function has multiple return values, they are also positional. That is, the first output parameter gets assigned to the first return value, and so on.
- If the function has multiple return values, the rules and restrictions are the same as for declaration functions. For more information, see "Multiple Return Values" on page 3-22.
- The `definitions` section can contain intermediate terms and `if-elseif-else` statements. The same syntax rules as in the declaration section of a `let` statement apply. For more information, see "Using Intermediate Terms in Equations" on page 2-35.
- The `definitions` section cannot contain expressions with dynamic semantics, such as `integ`, `time`, `der`, `edge`, `initialevent`, or `delay`.

Packaging Rules

- Simscape function files can reside directly on MATLAB path or in package directories. For more information, see "Organizing Your Simscape Files" on page 4-25.
- You can use source protection, as described in "Using Source Protection for Simscape Files" on page 4-26.
- Importing a package imports all the Simscape functions in this package. For more information, see "Importing Domain and Component Classes" on page 2-108.
- If a MATLAB function and a Simscape function have the same name, the MATLAB function has higher precedence.

Using Main Simscape Functions

The purpose of main Simscape functions is to reuse expressions in equations of multiple components, as well as in member declarations of domain or component files.

For example, exponential diode equations often use an expression that is a modification of `exp(`*`i`*`)`, to provide protection for large magnitudes of *i*. For details, see Diode and NPN Bipolar Transistor block reference pages. The "Simscape Functions" example shows how you can write a Simscape function to reuse this expression, instead of repeating it in every block:

```
function out = userFunction(x,y,z)
definitions
    out = if x > y
        (x-z)*exp(y);
    elseif x < -z
        (x+y)*exp(-z);
    else
        exp(x)
    end
end
end
```

Then, the Diode block can call this function with *y* and *z* values of `80` and `79`, respectively:

```
equations
   o == SimscapeFunction.Use.Functions.userFunction(i,80,79);
end
```

and the NPN Bipolar Transistor block can call the same function with values of `40` and `39`:

```
equations
   o == SimscapeFunction.Use.Functions.userFunction(i,40,39);
end
```

Recommended Ways of Code Reuse

Simscape language has a variety of tools that facilitate code reuse. Simscape functions and declaration functions let you reuse expressions. Subclassing and composite components let you reuse equations.

To reuse expressions across multiple components:

- Use main Simscape functions to reuse expressions in equations and member declarations.
- Use declaration functions in member declarations to reuse expressions that are out of Simscape expression capability. For more information, see "Declaration Functions" on page 3-22.

Functionality	Authoring Language	File extension	Usage	Supports Arguments with Units
Simscape function	Simscape	.ssc or .sscp	Member declaration and equations	Yes
Declaration function	MATLAB	.m or .p	Member declaration only	No

To reuse equations across multiple components:

- Use subclassing to model the "is-a" relationship between the base component and the derived component. The equations in the base component are reused in the derived component. For more information, see "Subclassing and Inheritance" on page 2-106.
- Use composite components to model the "has-a" relationship between the container component and the subcomponents. The equations in the member components are reused in the composite component. For more information, see "About Composite Components" on page 2-58.

Local Simscape Functions

Local Simscape functions reside inside a Simscape file that defines a component, domain, or another function, and are accessible only by that component, domain, or main function. For example, when you need to use a function in a single component only, defining it as a local function:

- Reduces the overhead of creating and packaging separate files.
- Restricts access, to ensure that only that specific component can use this function.

Include the local Simscape function in a component, domain, or function file, after the final `end` keyword that concludes the description of the component, domain, or main function. For example, this `spring` component uses a local function to modify its torque equation:

```
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
  branches
    t : r.t -> c.t; % torque through from node r to node c
  end
  equations
    assert(k>0)     % spring rate must be greater than zero
    w == r.w - c.w; % velocity across between node r and node c
    t == localTorque(k,theta);
    w == theta.der;
  end
end

function out = localTorque(in1,in2)
   definitions
      out = (in1*in2)*9.76 + 0.25; % Modification made to torque relationship
   end
end
```

The syntax rules for local functions are the same as for main functions. See “Syntax Rules” on page 3-25.

You can have multiple local functions declared in the same component, domain, or function file.

Local functions can contain calls to other local functions, in the same file, or to main functions in other files. In case of a name conflict, local functions have higher call precedence than main functions.

See Also

`function`

Related Examples

- “Simscape Functions”

More About

- “Declaration Functions” on page 3-22

Component Arrays

In this section...

About Component Arrays

Component arrays provide an intuitive way to model composite components with an arbitrary number of homogeneous members, such as segmented pipelines, battery packs, or transmission lines.

Use a `for` loop to declare an array of member components:

```
for i=1:array_size
   components (ExternalAccess=none)
  member_comp(i) = compX;
   end
end
```

The array size can be declared as an adjustable parameter of the composite component, so that the block users can modify this value.

```
parameters
   array_size = 10;               % Number of member components
end
```

Declare the parameter specifying the array size as a unitless integer because a `for` loop iterator must be a unitless integer.

Similar to regular composite components, if you want certain parameters of the underlying member component to be adjustable through the composite component interface, include them in the member declaration. This example establishes the relationship between parameter `parX` of the member component `compX` and the top-level parameter `top_level_parX` of the composite component:

```
parameters
   array_size = 10;              % Number of member components
   top_level_parX = { 1, 'm' };  % Modifiable parameter of the member components
end
for i=1:array_size
   components (ExternalAccess=none)
  member_comp(i) = compX(parX = top_level_parX);
   end
end
```

Use `for` loops to specify connections between the member components. The iterator range for these `for` loops depends on the array size and the type of connection. For example, when you connect `N` members in parallel, the iterator range is equal to the array size:

```
for i=1:N
   connections
  connect(compX(i).A, A);
  connect(compX(i).B, B);
   end
end
```

However, if you connect `N` members in series, the iterator range is from `1` to `(N-1)`, because you are connecting port **B** of each member except the last one to port **A** of the next member:

```
for i=1:(N-1)
   connections
```

```
    connect(compX(i).B, compX(i+1).A);
     end
  end
```

In this case, do not forget to connect the ends of the chain to the external ports of the composite component:

```
  connections
     connect(compX(1).A, A);
     connect(compX(N).B, B);
  end
```

You can also use `compX(end)` to indicate the last member of a component array. For example, this syntax is equivalent to the previous one, for connecting the ends of the chain to the external ports of the composite component:

```
  connections
     connect(compX(1).A, A);
     connect(compX(end).B, B);
  end
```

You can use nested `for` loops to create multidimensional arrays of components.

Syntax Rules and Restrictions

The following rules and restrictions apply to arrays in Simscape language:

- Arrays apply only to the `component` member class.
- Component arrays must be homogeneous, that is, their members must all belong to the same class. However, members can have different parameter values.
  - For an example of a component array with identical members, see “Segmented Pipeline Using Component Array” on page 3-32.
  - For an example of how you can specify different parameter values for certain members of a component array, see “Case Study — Battery Pack with Fault Using Arrays” on page 3-34.
- The array size can be a parameter, or a parametric expression. Parameters that control the array size can have their `ExternalAccess` attribute set to `modify`, which enables the block users to change the size of the array.
- Array members must have the `ExternalAccess` attribute set to `none`. This means that data logging, Variable Viewer, and other functionality that requires `ExternalAccess` to be `modify` or `observe` is not available to array members.
- Empty arrays are not supported.

You can use `for` loops to declare component arrays and to connect members of the array to each other. `for` loops have the same syntax as `for` in MATLAB. The following rules and restrictions apply to `for` loops in Simscape language:

- `for` loops can contain only `components` or `connections`.
- The `for` loop iterator must be a unitless integer.
- `for` loops can be nested. Use nested `for` loops to create multidimensional arrays of components.
- In nested `for` loops, the iterator in a nested loop cannot refer to an iterator in a loop above it. For example, this syntax is invalid:

  ```
  for i=1:N
     for j=1:i
  ```

```
      ...
    end
end
```

- Component declaration using a `for` loop must contain the `for` loop iterator on the left side as a bare identifier, for example, `pipe(i)`. You cannot use expressions or numbers in place of an iterator. Components declared inside a nested `for` loop must list all the iterators, for example:

```
for i=1:N
   for j=1:M
     components (ExternalAccess=none)
    resistor(i,j) = foundation.electrical.elements.resistor(R = R);
     end
   end
end
```

- You cannot include conditional sections (that you use to define component variants) inside `for` loops. However, you can include `for` loops inside the conditional sections.

See Also

More About

- “Segmented Pipeline Using Component Array” on page 3-32
- “Case Study — Battery Pack with Fault Using Arrays” on page 3-34

Segmented Pipeline Using Component Array

This example shows how you can model a segmented pipeline using a component array. This segmented pipeline model is a composite component that consists of *N* identical segments connected in series. Individual pipe segments are represented by the Pipe (IL) blocks from the Foundation library. N is a parameter that the block user can modify.

```
component SegmentedPipeline
  parameters
     N = 10;             % Number of segments
     segm_length = { 5, 'm' };     % Length of each segment
  end

  % Ports at the two ends of the pipeline
  nodes
     A = foundation.isothermal_liquid.isothermal_liquid; % A:left
     B = foundation.isothermal_liquid.isothermal_liquid; % B:right
  end

  % Declare array of N components
  for i=1:N
     components (ExternalAccess=none)
    pipe(i) = foundation.isothermal_liquid.elements.pipe(length = segm_length);
     end
  end

  % Connect all segments in series
  for i=1:(N-1)
     connections
    connect(pipe(i).B, pipe(i+1).A);
     end
  end

  % Connect two ends of pipeline to first and last segment, respectively
  connections
     connect(A, pipe(1).A);
     connect(B, pipe(N).B);
  end
end
```

In this example, for the sake of simplicity, the `SegmentedPipeline` component has only two modifiable parameters: `N` (**Number of segments**) and `segm_length` (**Length of each segment**). However, you can make other parameters of the underlying Pipe (IL) block accessible from the top-level composite component block dialog, as described in “Parameterizing Composite Components” on page 2-60. Parameter `N`, which defines the array size and is going to be used as the upper limit for the `for` loop iterator, is declared as a unitless integer.

Use a `for` loop to declare an array of `N` member components:

```
  for i=1:N
     components (ExternalAccess=none)
    pipe(i) = foundation.isothermal_liquid.elements.pipe(length = segm_length);
     end
  end
```

In this example, all the pipe segments have the same length. For an example of array component members having different parameter values, see “Case Study — Battery Pack with Fault Using Arrays” on page 3-34.

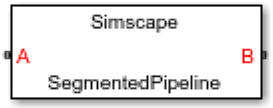
Use another `for` loop to connect all segments in series, by connecting node B of each pipe segment (except the last one) to node A of the next segment:

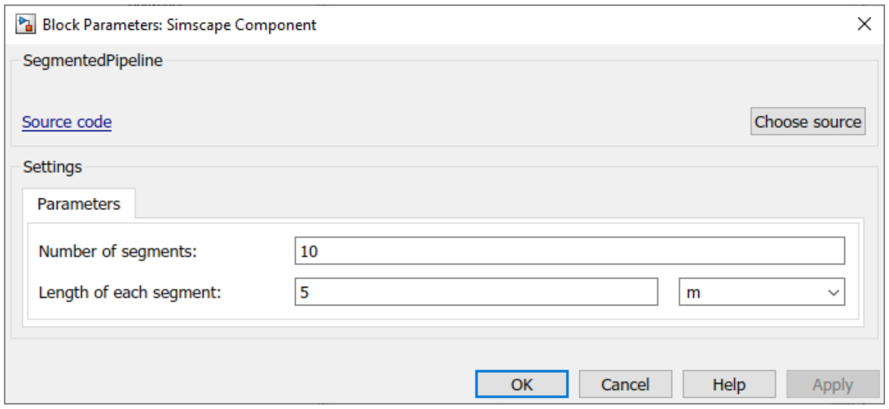
```
  for i=1:(N-1)
     connections
    connect(pipe(i).B, pipe(i+1).A);
     end
  end
```

Finally, connect the internal chain of segments to the two ends of pipeline, by connecting node A of the composite component to node A of the first segment and connecting node B of the composite component to node B of the last segment:

```
  connections
     connect(A, pipe(1).A);
     connect(B, pipe(N).B);
  end
end
```

The resulting block has two isothermal liquid ports, **A** and **B**, and two modifiable parameters: **Number of segments** and **Length of each segment**.




Block Parameters: Simscape Component

SegmentedPipeline

Source code

Settings

Parameters

Number of segments: 10

Length of each segment: 5 m

See Also

More About

- “Component Arrays” on page 3-29
- “Case Study — Battery Pack with Fault Using Arrays” on page 3-34

Case Study — Battery Pack with Fault Using Arrays

In this section...

Overview of the Model

This case study explains how you can use component arrays to model a battery pack consisting of multiple series-connected cells. It also shows how you can introduce a fault into one of the cells to see the impact on battery performance and cell temperatures. Both the number of cells and the position of the faulted cell are the top-level component parameters modifiable by the block user.

The case study is based on the "Lithium-Ion Battery Pack With Fault Using Arrays" example. To open the example model, type `ssc_lithium_battery_arrays` in the MATLAB Command Window.

The Battery Pack block is a composite component modeling an array of battery cells. The source files for this example are in the following package folder:

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

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

```
matlabroot
```

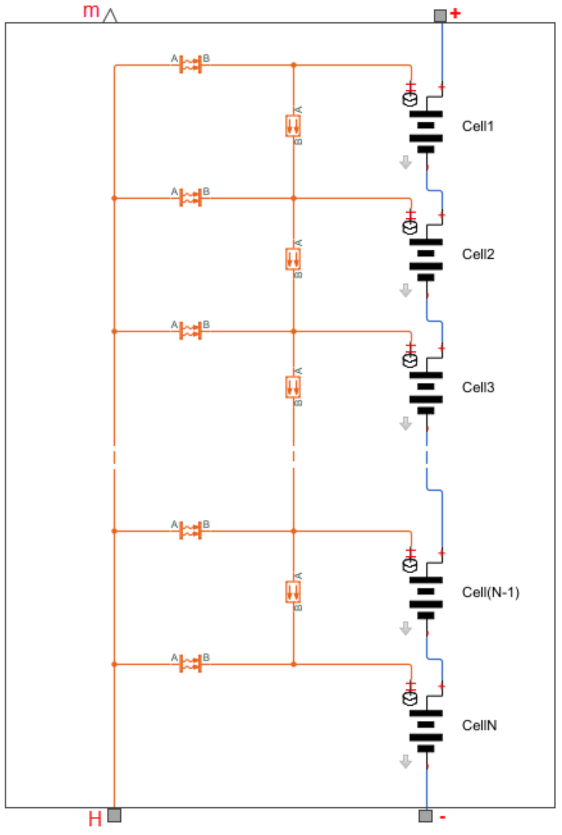
in the MATLAB Command Window.

The `+BatteryPack` package contains the following files:

- `battery_cell.ssc` — Component file representing the individual battery cell. The source for this component is generated using `subsystem2ssc` from the Lithium Cell 1RC subsystem in the "Lithium Battery Cell - One RC-Branch Equivalent Circuit" example.
- `battery_pack.ssc` — Composite component that models the battery pack as an array of `battery_cell` components.

```
component battery_pack
% Battery Pack
% This block models a scalable battery pack with faults using arrays.

% Copyright 2019 The MathWorks, Inc.

parameters
    Ncells = 20; % Number of series-connected cells
    cell_mass = {1, 'kg'}; % Cell mass
    cell_area = {0.1019, 'm^2'}; % Cell area
    h_conv    = {5, 'W/(m^2 * K)'}; % Heat transfer coefficient
    cell_Cp_heat = {810.5328, 'J/(kg*K)'}; %Cell specific heat
    Qe_init = {15.6845, 'hr*A'}; %Initial cell charge deficit
    T_init = {293.15, 'K'}; % Initial cell temperature
    SOC_LUT = [0; .1; .25; .5; .75; .9; 1]; %SOC table breakpoints (Mx1 array)
    Temperature_LUT = {[278.15, 293.15, 313.15], 'K'}; %Temperature table breakpoints (1xN array)
    Capacity_LUT = {[28.0081, 27.625, 27.6392], 'hr*A'}; %Capacity (1xN table)
    Em_LUT = {[3.4966, 3.5057, 3.5148; 3.5519, 3.566, 3.5653; 3.6183, 3.6337, 3.6402; 3.7066, 3.7127, 3.7213; 3.9131, 3.9259, 3.9376; 4.
    R0_LUT = {[.0117, .0085, .009; .011, .0085, .009; .0114, .0087, .0092; .0107, .0082, .0088; .0107, .0083, .0091; .0113, .0085, .0089
    R1_LUT = {[.0109, .0029, .0013; .0069, .0024, .0012; .0047, .0026, .0013; .0034, .0016, .001; .0033, .0023, .0014; .0033, .0018, .00
    C1_LUT = {[1913.6, 12447, 30609; 4625.7, 18872, 32995; 23306, 40764, 47535; 10736, 18721, 26325; 18036, 33630, 48274; 12251, 18360,
    % Fault cell
    fault_cell_position = 10; %Fault cell position
    fault_cell_Capacity_LUT = {[28.0081, 27.625, 27.6392]*0.95, 'hr*A'}; %Fault cell capacity (1xN table)
    fault_cell_Em_LUT = {[3.4966, 3.5057, 3.5148; 3.5519, 3.566, 3.5653; 3.6183, 3.6337, 3.6402; 3.7066, 3.7127, 3.7213; 3.9131, 3.9259,
    fault_cell_R0_LUT = {[.0117, .0085, .009; .011, .0085, .009; .0114, .0087, .0092; .0107, .0082, .0088; .0107, .0083, .0091; .0113, .
    fault_cell_R1_LUT = {[.0109, .0029, .0013; .0069, .0024, .0012; .0047, .0026, .0013; .0034, .0016, .001; .0033, .0023, .0014; .0033,
    fault_cell_C1_LUT = {[1913.6, 12447, 30609; 4625.7, 18872, 32995; 23306, 40764, 47535; 10736, 18721, 26325; 18036, 33630, 48274; 122
end

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

variables(Access=protected)
    T = {ones(1,Ncells),'K'};
    SOC = ones(1,Ncells);
end

outputs
    m = {ones(1,Ncells),'K'}; % m:top
end

for i =1:Ncells
    components(ExternalAccess=none)
        battery_cell(i) = BatteryPack.battery_cell(cell_mass=cell_mass,cell_Cp_heat=cell_Cp_heat,...
            C1_LUT=(if i==fault_cell_position,fault_cell_C1_LUT;else C1_LUT; end),...
            SOC_LUT=SOC_LUT,Temperature_LUT=Temperature_LUT,...
            Capacity_LUT=(if i==fault_cell_position,fault_cell_Capacity_LUT;else Capacity_LUT; end),...
            Em_LUT=(if i==fault_cell_position,fault_cell_Em_LUT;else Em_LUT; end),Qe_init=Qe_init,...
            R0_LUT=(if i==fault_cell_position,fault_cell_R0_LUT;else R0_LUT; end),...
            R1_LUT=(if i==fault_cell_position,fault_cell_R1_LUT;else R1_LUT; end),T_init.value=T_init);
        convection(i) = foundation.thermal.elements.convection(area=cell_area,heat_tr_coeff=h_conv);
    end

    connections
        connect(battery_cell(i).H,convection(i).B);
        connect(H,convection(i).A);
    end
end

connections
    connect(battery_cell(1).p,p);
    connect(battery_cell(Ncells).n,n);
end

equations
    assert(mod(Ncells, 1) == 0 && Ncells > 0, 'Number of series-connected cells must be a positive integer');
    assert(mod(fault_cell_position, 1) == 0 && fault_cell_position > 0 && fault_cell_position <= Ncells , 'Fault cell position must be a
    T == [battery_cell.T_init];
    SOC == [battery_cell.SOC];
    m == T;
end

for i=1:Ncells-1
    components(ExternalAccess=none)
        conduction(i) = foundation.thermal.elements.conduction(area={1e-3,'m^2'},...
            th_cond={200,'W/(m*K)'});
    end
    connections
        connect(battery_cell(i+1).p,battery_cell(i).n);
        connect(battery_cell(i).H,conduction(i).A);
        connect(battery_cell(i+1).H,conduction(i).B);
    end
end

end
```

This schematic represents the equivalent circuit for the composite component.



The composite component has two electrical nodes, `p` and `n`, and one thermal node, `H`:

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

The component also has an output, `m`, to output the temperature data:

```
outputs
    m = {ones(1,Ncells),'K'}; % m:top
end
```

Introducing the Fault

The fault is represented by changing the parameters for one of the battery cells, reducing both capacity and open-circuit voltage, and increasing the resistance values.

To account for the fault, top-level composite component parameters are divided in two groups: generic parameters and those specific to the faulted cell. For example, cell mass is the same for all cells. However, the capacitance of the faulted cell is different from all the other cells, therefore it needs a separate `fault_cell_Capacity_LUT` parameter, instead of the generic `Capacity_LUT`.

```
parameters
    Ncells = 20; % Number of series-connected cells
    cell_mass = {1, 'kg'}; % Cell mass
    cell_area = {0.1019, 'm^2'}; % Cell area
    h_conv    = {5, 'W/(m^2 * K)'}; % Heat transfer coefficient
    cell_Cp_heat = {810.5328, 'J/(kg*K)'}; %Cell specific heat
    Qe_init = {15.6845, 'hr*A'}; %Initial cell charge deficit
    T_init = {293.15, 'K'}; % Initial cell temperature
    SOC_LUT = [0; .1; .25; .5; .75; .9; 1]; %SOC table breakpoints (Mx1 array)
    Temperature_LUT = {[278.15, 293.15, 313.15], 'K'}; %Temperature table breakpoints (1xN array)
    Capacity_LUT = {[28.0081, 27.625, 27.6392], 'hr*A'}; %Capacity (1xN table)
    Em_LUT = {[3.4966, ... 4.1928, 4.193], 'V'}; %Em open-circuit voltage, Em (MxN table)
    R0_LUT = {[.0117, ... .0085, .0089], 'Ohm'}; %R0 terminal resistance (MxN table)
    R1_LUT = {[.0109, ... .0017, .0011], 'Ohm'}; %R1 cell resistance (MxN table)
    C1_LUT = {[1913.6, ... 23394, 30606], 'F'}; %C1 capacitance (MxN table)
    % Fault cell
    fault_cell_position = 10; %Fault cell position
    fault_cell_Capacity_LUT = {[28.0081, 27.625, 27.6392]*0.95, 'hr*A'}; %Fault cell capacity (1xN table)
    fault_cell_Em_LUT = {[3.4966, ... 4.1928, 4.193]*0.90, 'V'}; %Fault cell Em open-circuit voltage, Em (MxN table)
    fault_cell_R0_LUT = {[.0117, ... .0085, .0089]*5, 'Ohm'}; % Fault cell R0 terminal resistance (MxN table)
    fault_cell_R1_LUT = {[.0109, ... .0017, .0011]*5, 'Ohm'}; % fault cell R1 cell resistance (MxN table)
    fault_cell_C1_LUT = {[1913.6, ... 23394, 30606]*0.95, 'F'}; % Fault cell C1 capacitance (MxN table)
end
```

Both the number of cells, `Ncells`, and the position of the faulted cell, `fault_cell_position`, are top-level parameters of the composite component, which means that they will be modifiable by the block user.

Declaring Arrays of Member Components

Declare an array of cells, with the number of elements defined by the `Ncells` parameter.

```
for i =1:Ncells
    components(ExternalAccess=none)
        battery_cell(i) = BatteryPack.battery_cell(cell_mass=cell_mass,cell_Cp_heat=cell_Cp_heat,...
            C1_LUT=(if i==fault_cell_position,fault_cell_C1_LUT;else C1_LUT; end),...
            SOC_LUT=SOC_LUT,Temperature_LUT=Temperature_LUT,...
            Capacity_LUT=(if i==fault_cell_position,fault_cell_Capacity_LUT;else Capacity_LUT; end),...
            Em_LUT=(if i==fault_cell_position,fault_cell_Em_LUT;else Em_LUT; end),Qe_init=Qe_init,...
            R0_LUT=(if i==fault_cell_position,fault_cell_R0_LUT;else R0_LUT; end),...
            R1_LUT=(if i==fault_cell_position,fault_cell_R1_LUT;else R1_LUT; end),T_init.value=T_init);
    end
end
```

For each member, associate its parameters with the top-level parameters of the composite component. For the iterator value that corresponds to the faulted cell position, specify the faulted cell parameters instead of the respective generic ones, as needed. For example, all cells have the same cell mass:

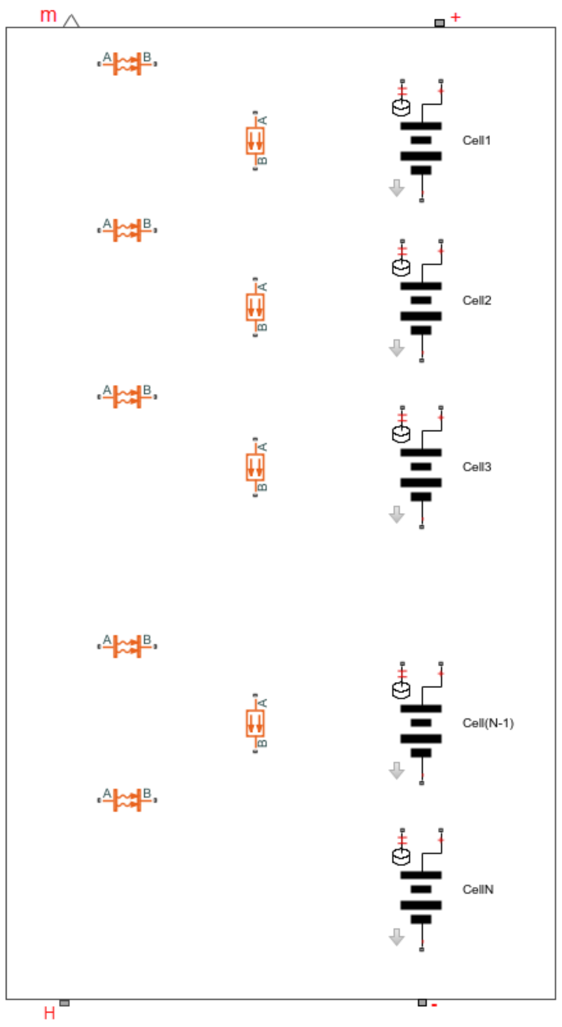
```
cell_mass=cell_mass
```

However, the capacitance of the faulted cell is different from all the other cells, therefore, for the iterator value that corresponds to the faulted cell position, assign the `Capacity_LUT` parameter of the cell to the `fault_cell_Capacity_LUT` parameter of the composite component, and for all other cells assign it to `Capacity_LUT`:

```
Capacity_LUT=(if i==fault_cell_position,fault_cell_Capacity_LUT;else Capacity_LUT; end)
```

An array of *N* cells requires *N* identical convections and *N-1* identical conductions. These are thermal components from the Foundation library.

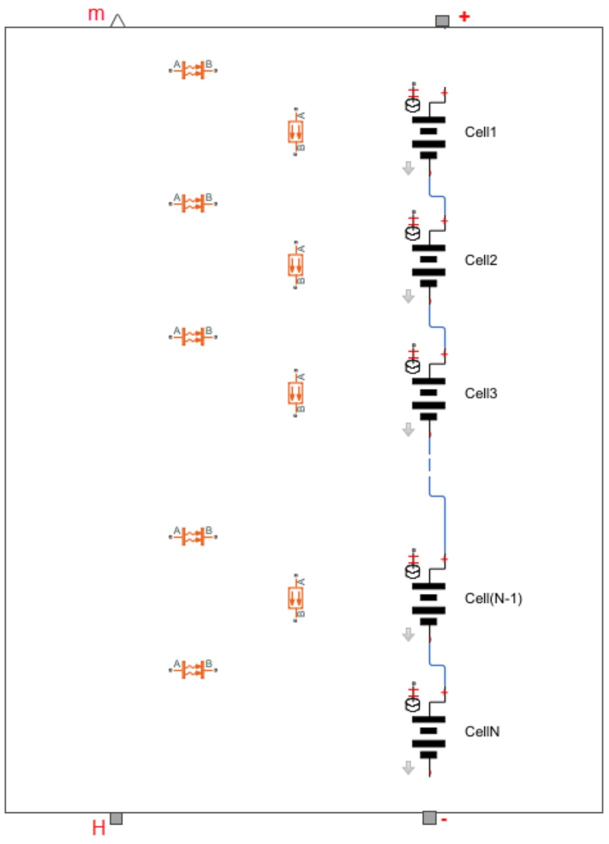
```
for i =1:Ncells
    components(ExternalAccess=none)
        convection(i) = foundation.thermal.elements.convection(area=cell_area,heat_tr_coeff=h_conv);
    end
end

for i=1:Ncells-1
    components(ExternalAccess=none)
        conduction(i) = foundation.thermal.elements.conduction(area={1e-3,'m^2'},...
            th_cond={200,'W/(m*K)'});
    end
end
```



Connecting the Components

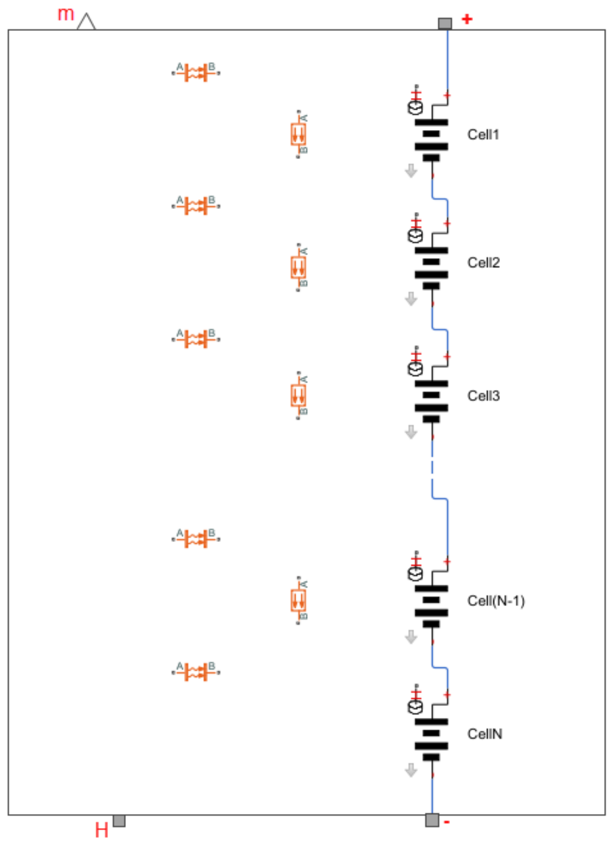
Use a `for` loop to connect all the electrical nodes of the cells in series, by connecting the negative port of each cell (except the last one) to the positive port of the next cell:

```
for i=1:Ncells-1
   connections
  connect(battery_cell(i+1).p,battery_cell(i).n);
   end
end
```



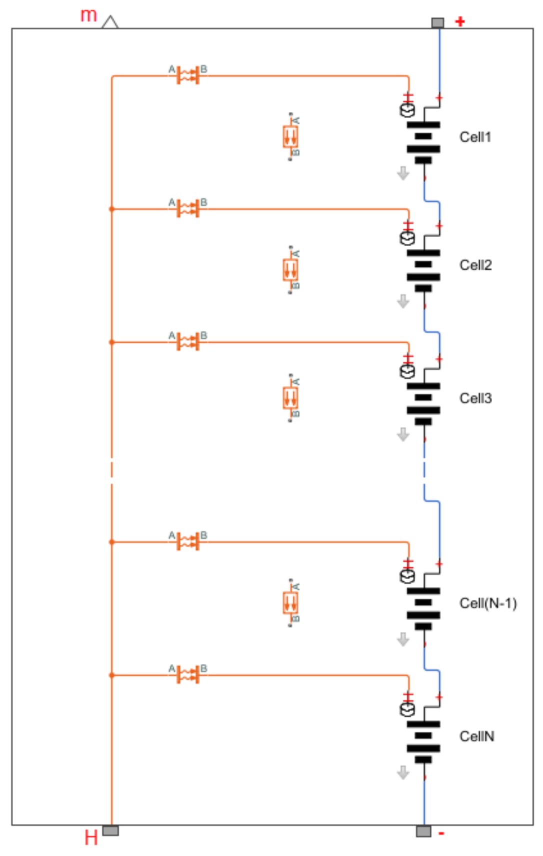
Then connect this internal chain to the two electrical nodes of the composite component:

```
connections
   connect(battery_cell(1).p,p);
   connect(battery_cell(Ncells).n,n);
end
```



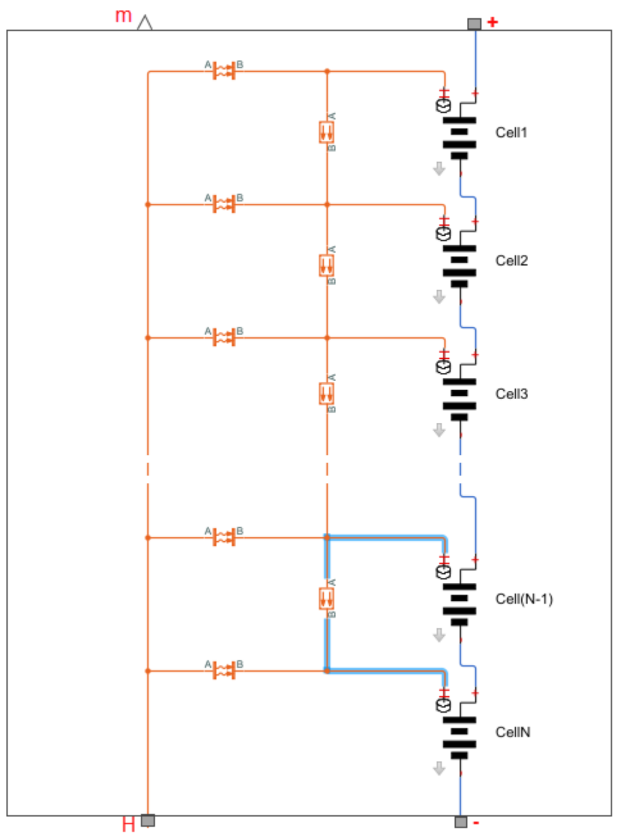
Use another `for` loop to connect thermal node `H` of each of the cells to node `B` of its respective convection component, and also to connect nodes `A` of all the convection components to thermal node `H` of the composite component:

```
for i=1:Ncells
   connections
     connect(battery_cell(i).H,convection(i).B);
     connect(H,convection(i).A);
   end
end
```



Finally, incorporate the thermal conduction elements by connecting node A to node H of the previous cell and node B to node H of the next cell:

```
for i=1:Ncells-1
   connections
      connect(battery_cell(i).H,conduction(i).A);
      connect(battery_cell(i+1).H,conduction(i).B);
   end
end
```



Outputting Data as a Numeric Array

The physical signal output port, `m`, outputs the temperature data for the battery pack as a multidimensional physical signal, composed of the temperature data for each cell. The vector size is determined by the `Ncells` parameter value.

```
outputs
    m = {ones(1,Ncells),'K'}; % m:top
end
```

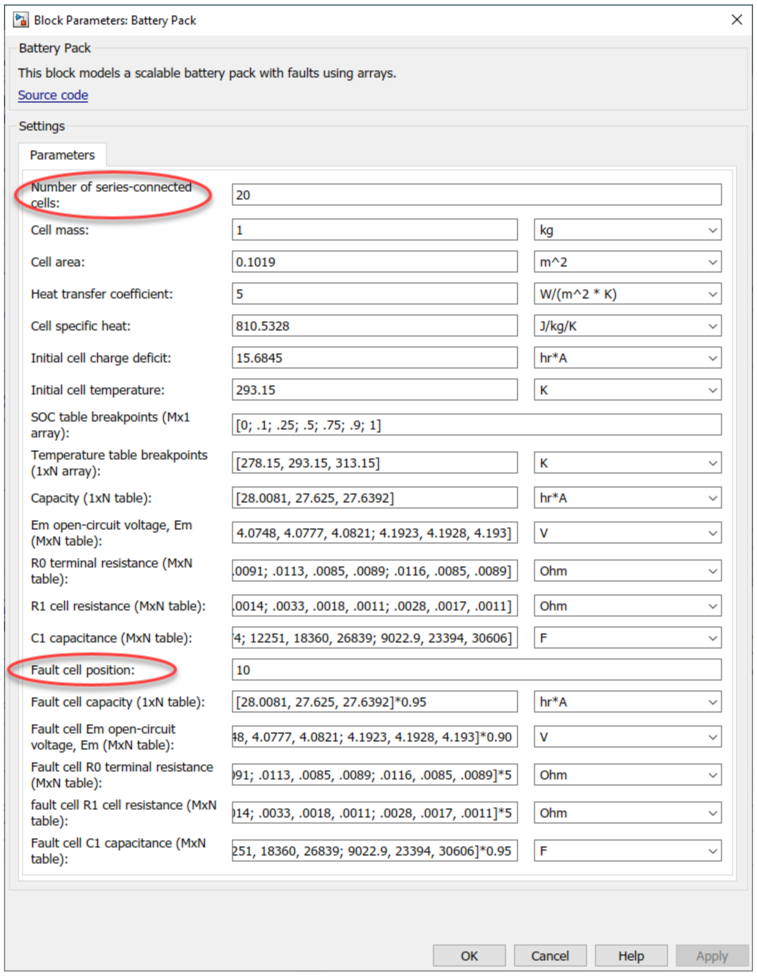
To get the temperature data from the array of components, use the `equations` section. Declare the variable `T` as protected, because public variables cannot reference parameters, and then use this variable to extract the temperature values from the member cells.

```
variables(Access=protected)
    T = {ones(1,Ncells),'K'};
end
equations
    T == [battery_cell.T_init];
    m == T;
end
```

Indexing into member component variables, `battery_cell.T_init`, returns a comma-separated list. You can then concatenate the comma-separated list into a numeric array, `[battery_cell.T_init]`, and use it in the component equations. For more information, see “How to Use the Comma-Separated Lists”.

Battery Pack Block

The Battery Pack block generated from this composite component has two electrical ports, **+** and **-**, a thermal port, **H**, and a physical signal output port, **m**. The parameters modifiable through the block interface include the number of cells in the battery pack and the position of the faulted cell. The block user can modify the parameters of the faulted cell to see the impact of the fault on battery performance and cell temperatures.



Block Parameters: Battery Pack

Battery Pack

This block models a scalable battery pack with faults using arrays.

Source code

Settings

Parameters

Number of series-connected cells:	20	
Cell mass:	1	kg
Cell area:	0.1019	m^2
Heat transfer coefficient:	5	W/(m^2 * K)
Cell specific heat:	810.5328	J/kg/K
Initial cell charge deficit:	15.6845	hr*A
Initial cell temperature:	293.15	K
SOC table breakpoints (Mx1 array):	[0; .1; .25; .5; .75; .9; 1]	
Temperature table breakpoints (1xN array):	[278.15, 293.15, 313.15]	K
Capacity (1xN table):	[28.0081, 27.625, 27.6392]	hr*A
Em open-circuit voltage, Em (MxN table):	4.0748, 4.0777, 4.0821; 4.1923, 4.1928, 4.193]	V
R0 terminal resistance (MxN table):	.0091; .0113, .0085, .0089; .0116, .0085, .0089]	Ohm
R1 cell resistance (MxN table):	.0014; .0033, .0018, .0011; .0028, .0017, .0011]	Ohm
C1 capacitance (MxN table):	4; 12251, 18360, 26839; 9022.9, 23394, 30606]	F
Fault cell position:	10	
Fault cell capacity (1xN table):	[28.0081, 27.625, 27.6392]*0.95	hr*A
Fault cell Em open-circuit voltage, Em (MxN table):	48, 4.0777, 4.0821; 4.1923, 4.1928, 4.193]*0.90	V
Fault cell R0 terminal resistance (MxN table):	91; .0113, .0085, .0089; .0116, .0085, .0089]*5	Ohm
fault cell R1 cell resistance (MxN table):	14; .0033, .0018, .0011; .0028, .0017, .0011]*5	Ohm
Fault cell C1 capacitance (MxN table):	251, 18360, 26839; 9022.9, 23394, 30606]*0.95	F

See Also

More About

- “Component Arrays” on page 3-29
- “Segmented Pipeline Using Component Array” on page 3-32

Simscape File Deployment

Generating Custom Blocks from Simscape Component Files

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. There are two mechanisms that let you do this:

- "Selecting Component File Directly from Block" on page 4-3 — Use the Simscape Component block, which you can find in the Utilities library, and point it to a Simscape component file. The block instantly acquires the properties based on the source component file: name, description, parameters, variables, appropriate ports and the custom icon image (if available). If you modify the underlying source file, the block reflects these changes. If you point the block to a different component file, the block properties change accordingly, to reflect the new source.

  Use this method to quickly deploy a single component file, to try out different variants of a component in your model, or to iterate on a component definition and get instant feedback.
- "Building Custom Block Libraries" on page 4-25 — Generate a custom block library from a package of Simscape component files. The package hierarchy determines the resulting library structure. You can customize the library name and appearance and provide annotation.

  Use this method to generate reusable custom block libraries.

See Also

Related Examples

- "Deploy a Component File in Block Diagram" on page 4-5
- "Switch Between Different Source Components" on page 4-9
- "Prototype a Component and Get Instant Feedback" on page 4-18
- "Create a Custom Block Library" on page 4-31
- "Customize Block Display" on page 4-45

More About

- "Customizing the Block Name and Appearance" on page 4-33
- "Customizing the Library Name and Appearance" on page 4-29

Selecting Component File Directly from Block

In this section...

Suggested Workflows

The Simscape Component block lets you select a Simscape component file, and then instantly acquires the properties based on that source component file: name, description, parameters, variables, the block icon and appropriate ports. For more information on how the component file elements translate into the properties of the block, see "Customizing the Block Name and Appearance" on page 4-33.

Use the Simscape Component block to:

- Quickly deploy a single Simscape component file as a block in your model, without the extra steps of packaging the file and building a custom library. For example, you wrote a component prototype yourself, got it from a colleague, or found it on MATLAB Central. Save the file in your current working directory, or anywhere on the MATLAB path, and use it as a source file for a Simscape Component block in your model. For more information on valid locations of a source component file, see "Component File Locations" on page 4-3. For an example of this workflow, see "Deploy a Component File in Block Diagram" on page 4-5.
- Try out different component implementations, to decide which implementation is most appropriate for your model. You can also use this workflow to test the differences between the old and new implementations of the same component. Instead of adding, deleting, and reconnecting different blocks in your model, you can use a single Simscape Component block and switch between the source component files. When you point a Simscape Component block to a different component file, the block properties change accordingly, to reflect the new source. For an example of this workflow, see "Switch Between Different Source Components" on page 4-9.
- Quickly try out different ideas for a physical component and get instant feedback on the resulting block implementation. This workflow lets you interactively modify the component source and immediately see the changes by refreshing the resulting block. For an example of this workflow, see "Prototype a Component and Get Instant Feedback" on page 4-18.

Component File Locations

When you deploy a component file by using the Simscape Component block, the component file does not have to be in a package. However, the directory where the file resides has to be on the MATLAB path. If the file resides in a package, then the package parent directory must be on the MATLAB path.

If you browse to a component file that is not on the path, then, when you try to select it, a File Not On Path dialog opens. Click **Add** to add the appropriate directory to the MATLAB path.

The **Add** button is similar to the `addpath` command, that is, it adds the folder to the path only for the duration of the current MATLAB session. If you do not save the path and then open the model in a subsequent session, the Simscape Component block becomes unresolved.

If the source component is located in the current working directory, then there is no requirement for it to be on the path. However, if you later try to open the model from another directory, the Simscape Component block also becomes unresolved.

It is good practice to keep the source component files that you want to reuse in a directory included in your permanent search path. For more information, see “What Is the MATLAB Search Path?”.

See Also

Related Examples

- “Deploy a Component File in Block Diagram” on page 4-5
- “Switch Between Different Source Components” on page 4-9
- “Prototype a Component and Get Instant Feedback” on page 4-18

More About

- “Customizing the Block Name and Appearance” on page 4-33

Deploy a Component File in Block Diagram

This example shows how you can quickly transform a Simscape component file into a block in your model, without the extra steps of packaging the file and building a custom library.

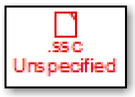
Suppose you have the following Simscape file, named `my_resistor`, in your working directory:

```
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' };     % Current
    v = { 0, 'V' };     % Voltage
  end
  parameters
    R = { 1, 'Ohm' };   % Resistance
  end

  branches
    i : p.i -> n.i;
  end

  equations
    assert(R>0)
    v == p.v - n.v;
    v == i*R;
  end

end
```

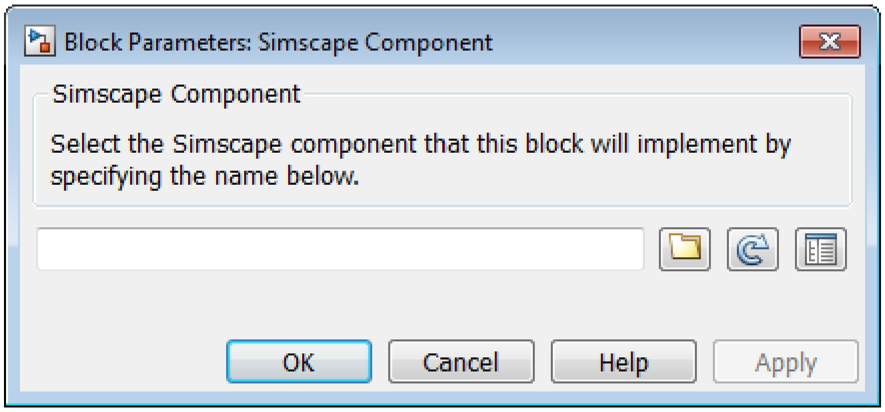
Tip This component implements a linear resistor. It is described in more detail in “Model Linear Resistor in Simscape Language” on page 1-3. You can copy the source from this page and save it as `my_resistor.ssc` in your working directory.

To deploy this component as a block in your model:

1. Open or create a model.
2. Open the Simscape > Utilities library and add the Simscape Component block to your model. At first, the block does not point to any component file. Therefore, it does not have any ports, and the block icon states it is `Unspecified`.



3. Double-click the block to open the source file selector dialog box.

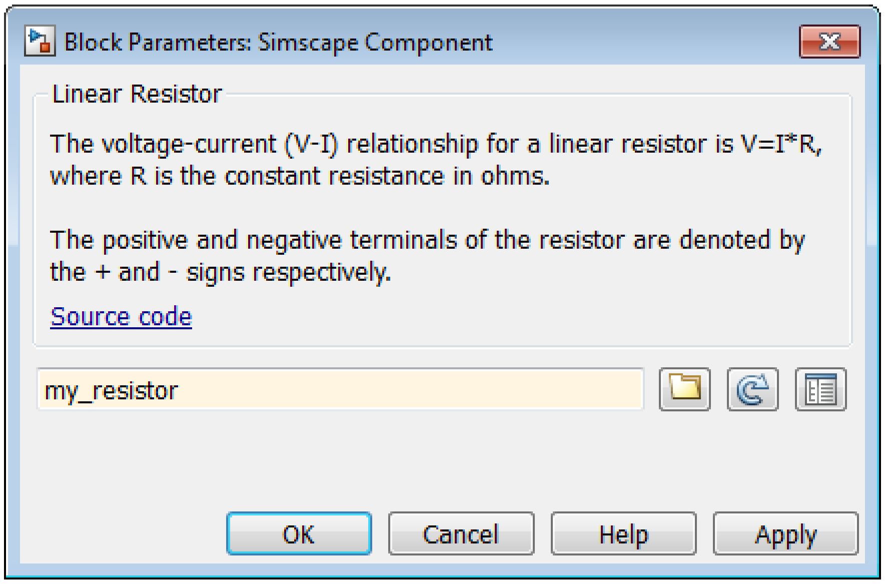


Block Parameters: Simscape Component

Simscape Component

Select the Simscape component that this block will implement by specifying the name below.

4 Click to open the browser. The browser opens in the current working directory and lists only the files with the `.ssc` or `.sscp` extension. Select the `my_resistor.ssc` file and click **Open**. The name of the source file appears in the text field of the source file selector dialog box, and the block name, description, and the link to source code appear in the preview pane.



Block Parameters: Simscape Component

Linear Resistor

The voltage-current (V-I) relationship for a linear resistor is V=I*R, where R is the constant resistance in ohms.

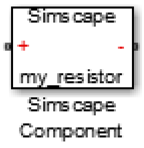
The positive and negative terminals of the resistor are denoted by the + and - signs respectively.

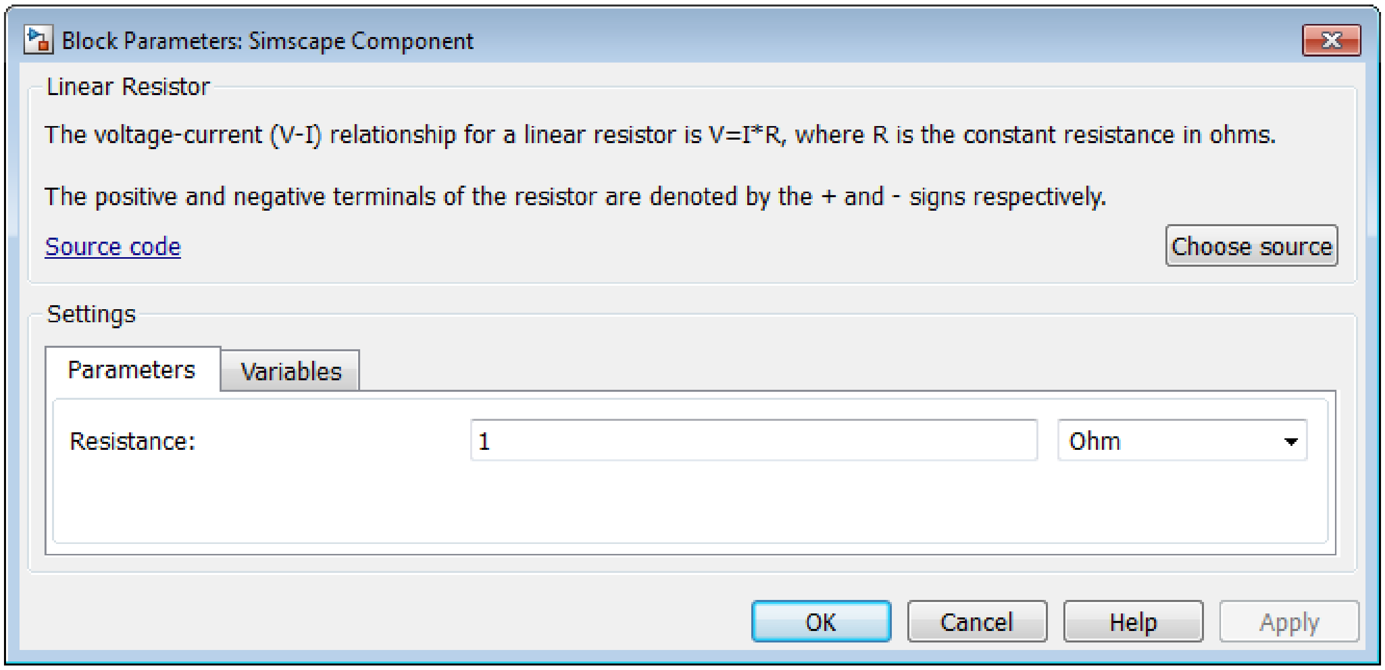
Source code

my_resistor

Tip Instead of browsing, you can type `my_resistor` directly into the text field. In this case, however, the preview pane does not automatically get updated. If you want to preview the block name, description, or source code, click .

5 Click **Apply**. The block icon and dialog box get updated, based on the selected source component.





Block Parameters: Simscape Component

Linear Resistor

The voltage-current (V-I) relationship for a linear resistor is V=I*R, where R is the constant resistance in ohms.

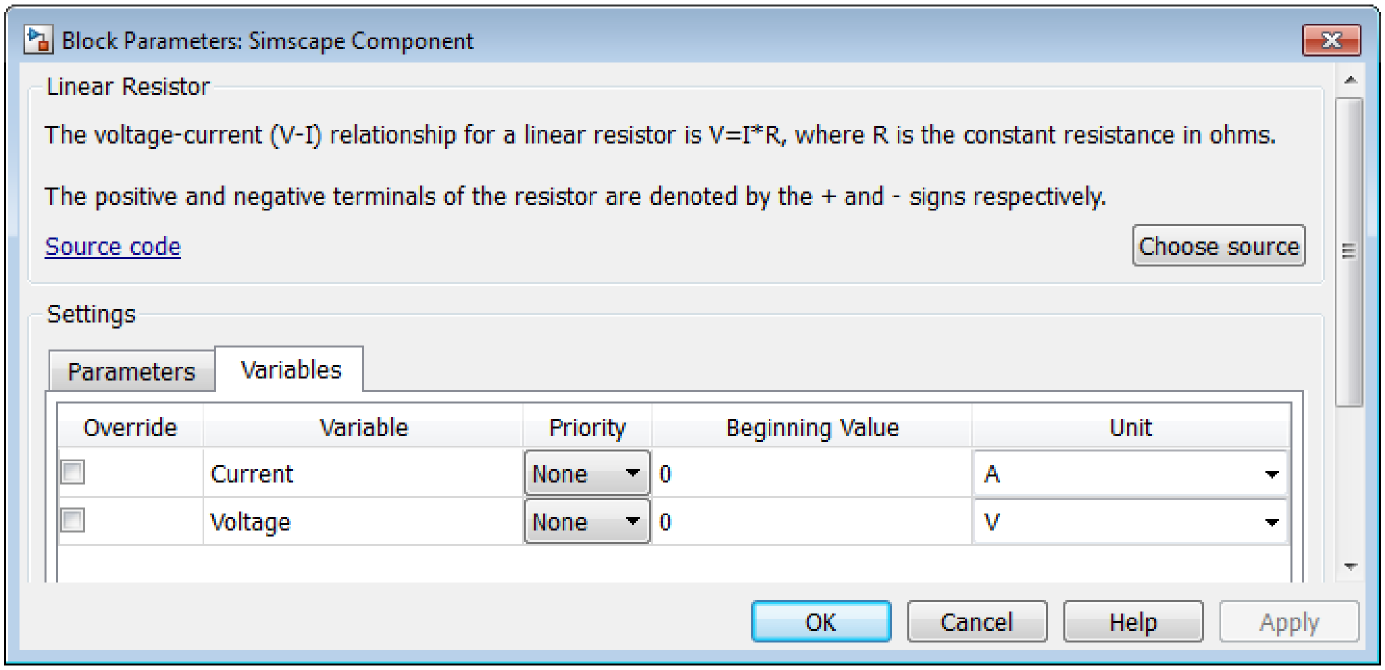
The positive and negative terminals of the resistor are denoted by the + and - signs respectively.

Source code

Settings

Parameters | Variables

Resistance: 1 Ohm



Block Parameters: Simscape Component

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.

Source code

Settings

Parameters | Variables

Override	Variable	Priority	Beginning Value	Unit
☐	Current	None	0	A
☐	Voltage	None	0	V

See Also

Related Examples

- “Model Linear Resistor in Simscape Language” on page 1-3
- “Customize Block Display” on page 4-45
- “Switch Between Different Source Components” on page 4-9

- “Prototype a Component and Get Instant Feedback” on page 4-18

More About

- “Selecting Component File Directly from Block” on page 4-3
- “Customizing the Block Name and Appearance” on page 4-33

Switch Between Different Source Components

This example shows how you can try out several variants of a component in your model by pointing the Simscape Component block to different component files.

The component files used in this example are capacitor models with different levels of fidelity, to allow exploration of the effect of losses and nonlinearity. The source files are part of your product installation, located 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
```

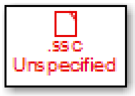
in the MATLAB Command Window. For more information about these capacitor models, see “Case Study — Basic Custom Block Library” on page 4-55.

To test capacitor models of different fidelity:

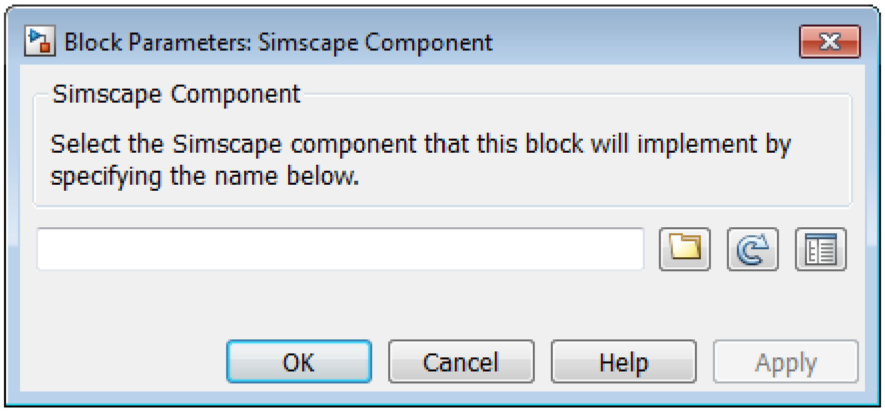
1 To create a new model with optimal settings for physical modeling, in the MATLAB Command Window, type:

```
ssc_new
```

2 Open the Simscape > Utilities library and add the Simscape Component block to your model. At first, the block does not point to any component file, therefore it does not have any ports and the block icon says `Unspecified`.



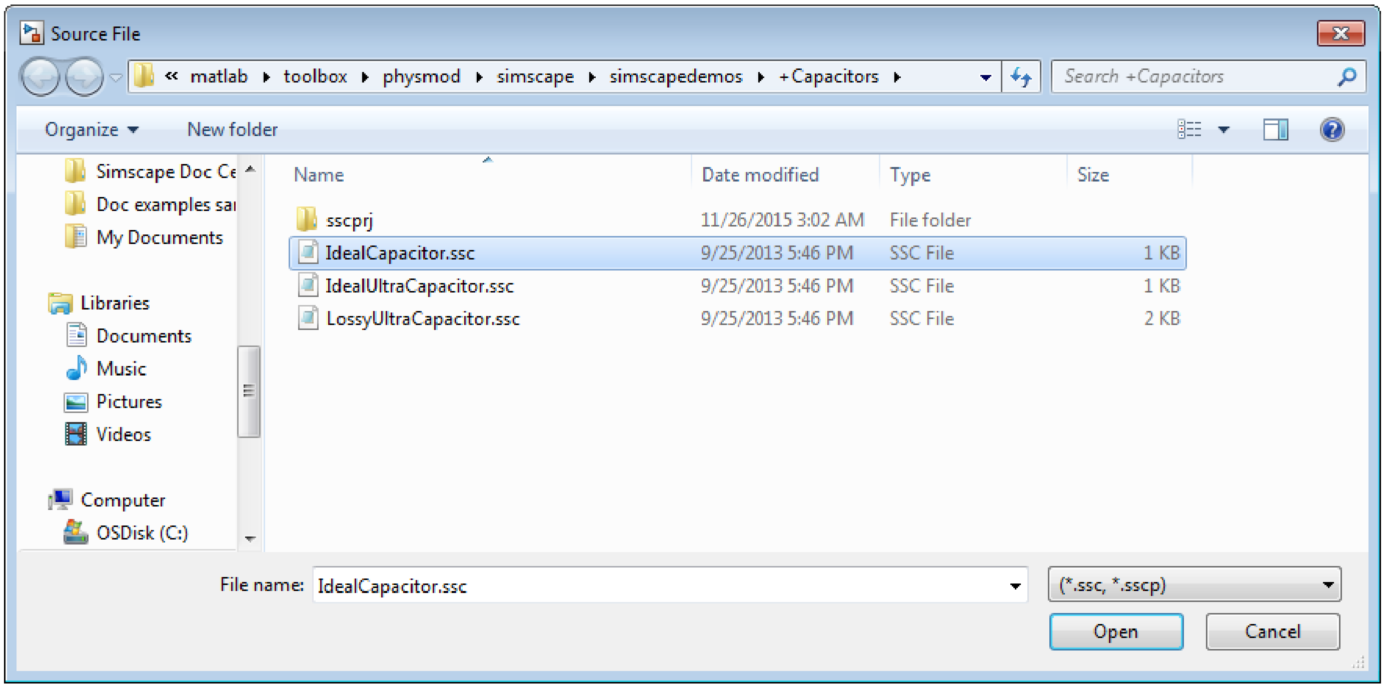
3 Double-click the block to open the source file selector dialog box.


Block Parameters: Simscape Component

Simscape Component

Select the Simscape component that this block will implement by specifying the name below.

4 Click 📁 and navigate to the directory containing the capacitor component files.

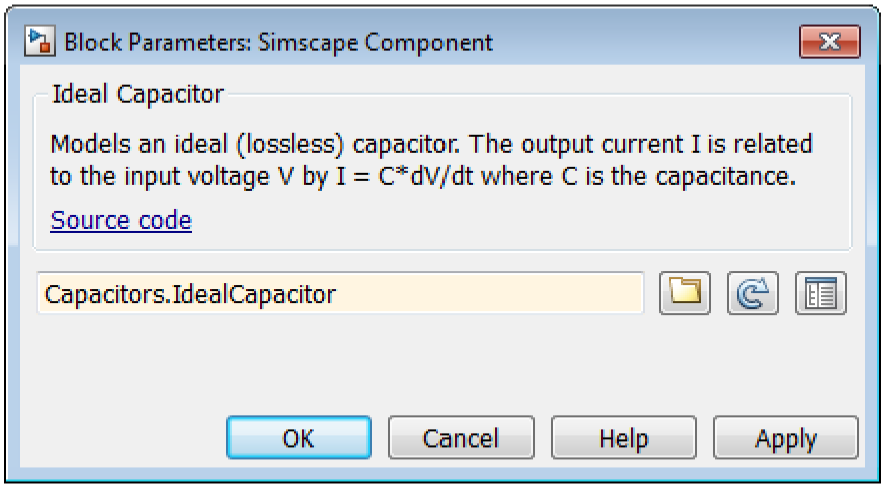


Name	Date modified	Type	Size
sscprj	11/26/2015 3:02 AM	File folder	
IdealCapacitor.ssc	9/25/2013 5:46 PM	SSC File	1 KB
IdealUltraCapacitor.ssc	9/25/2013 5:46 PM	SSC File	1 KB
LossyUltraCapacitor.ssc	9/25/2013 5:46 PM	SSC File	2 KB

File name: IdealCapacitor.ssc

(*.ssc, *.sscp)

5 Select the `IdealCapacitor.ssc` file and click **Open**. The name of the source file appears in the text field of the source file selector dialog box, and the block name, description, and the link to source code appear in the preview pane.



Ideal Capacitor

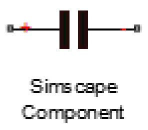
Models an ideal (lossless) capacitor. The output current I is related to the input voltage V by I = C*dV/dt where C is the capacitance.

Source code

Capacitors.IdealCapacitor

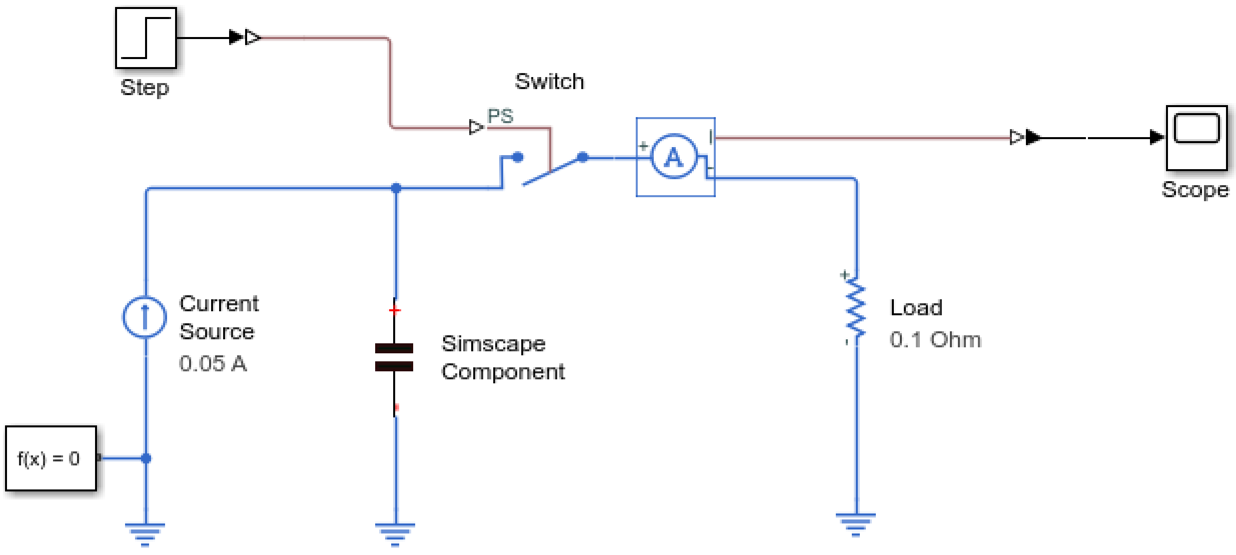
Note Because the component file resides in a package, the file name in the selector dialog box field is the full name, starting from the package root.

6 Click **OK**. The block icon gets updated, based on the selected source component.

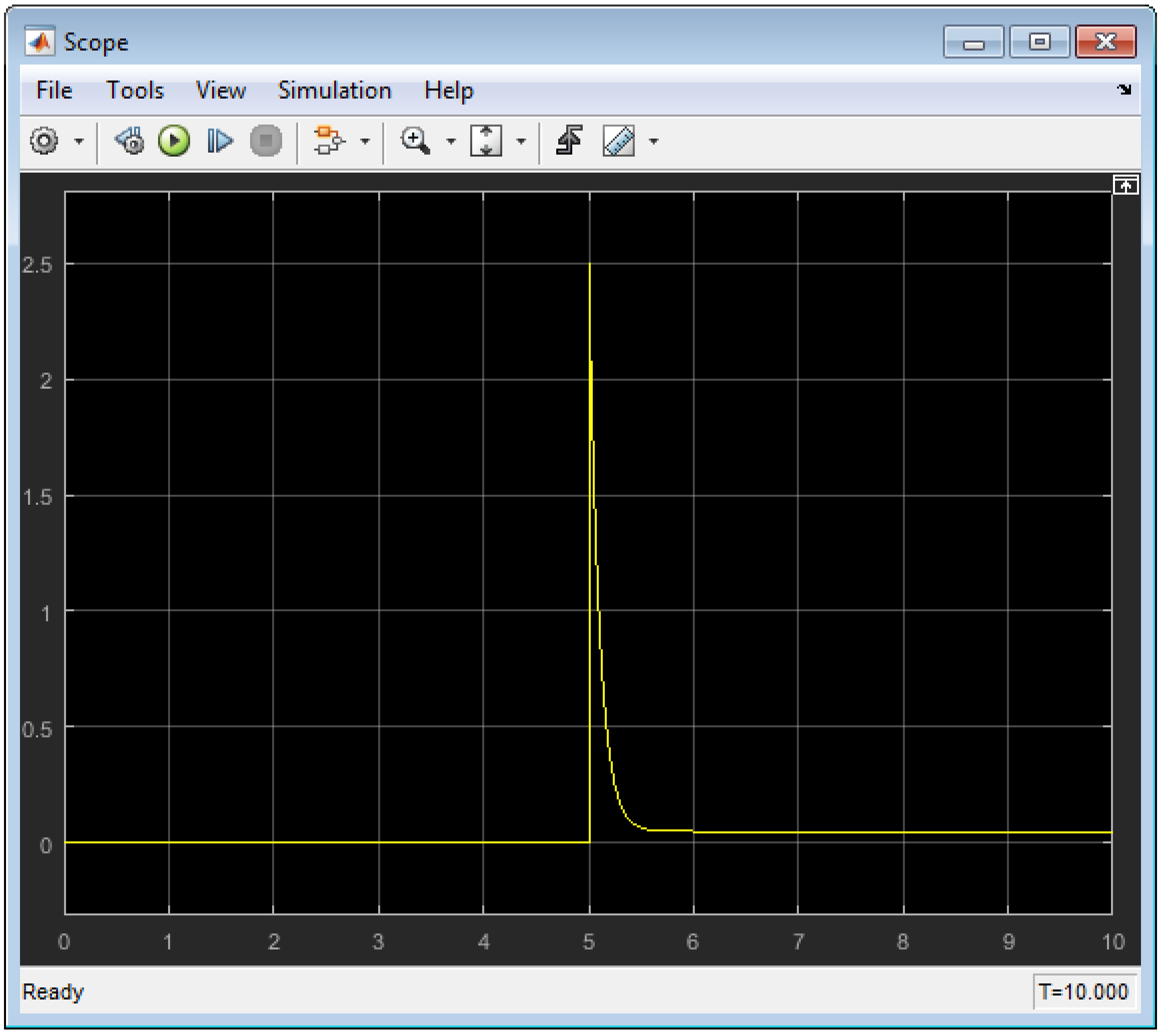


Note The `+Capacitors` package directory contains image files, with the names corresponding to the Simscape component files, that define customized block icons. Therefore, when you point the Simscape Component block to the `IdealCapacitor.ssc` source file, it uses the `IdealCapacitor.jpg` in the same directory as the block icon. For details, see "Customize the Block Icon" on page 4-42.

7 Build the test model and connect the blocks as shown in the following diagram.

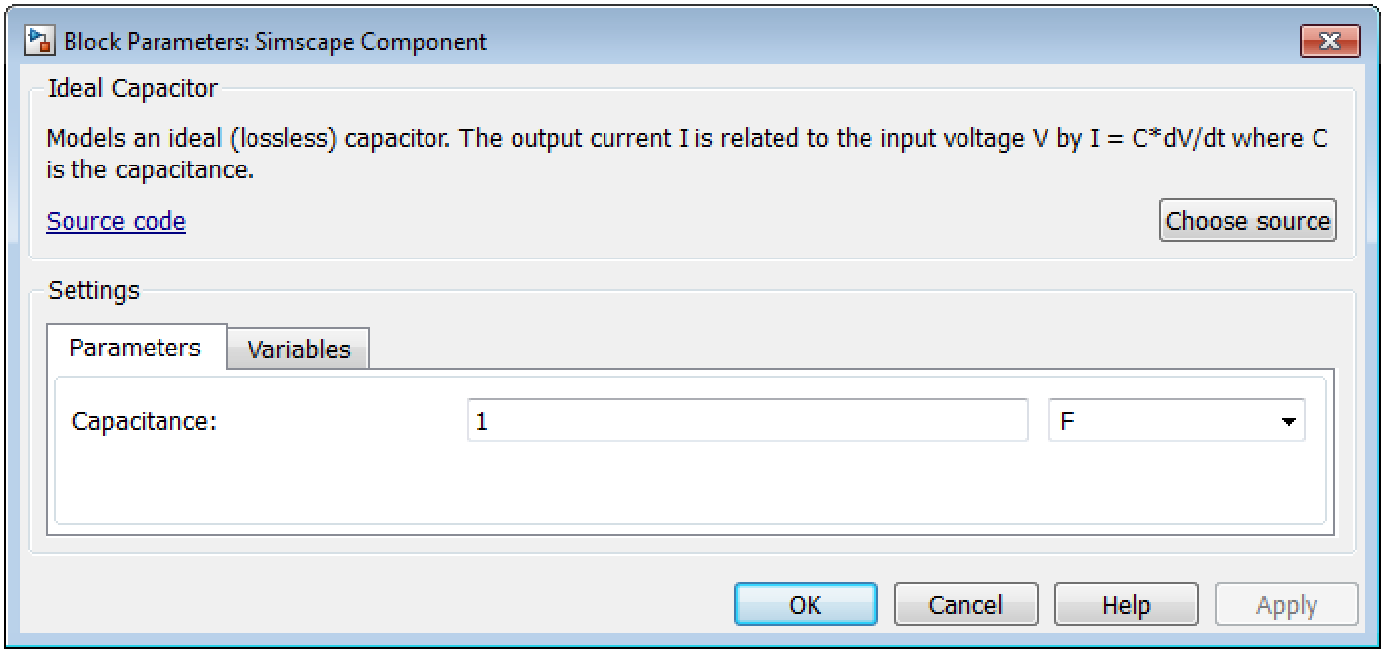


8 Open the scope and simulate the model.



The Simscape Component block points to an ideal capacitor component. Simulation results show that, when the switch is flipped at t=5 seconds, the capacitor delivers 2.5 A to the load.

9 To switch to another capacitor model, open the Simscape Component block dialog box and click **Choose source**.



Block Parameters: Simscape Component

Ideal Capacitor

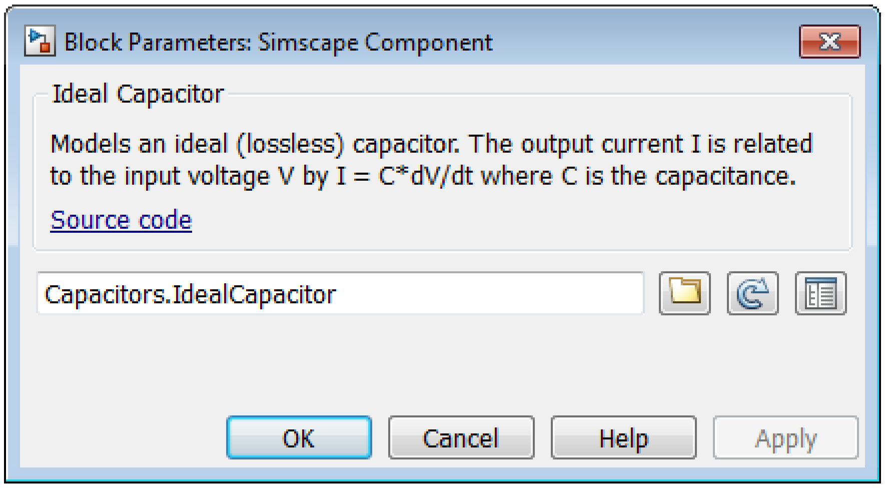
Models an ideal (lossless) capacitor. The output current I is related to the input voltage V by I = C*dV/dt where C is the capacitance.

Source code

Settings

Capacitance: 1 F

The source file selector dialog box opens, displaying the preview of the currently selected component.



Block Parameters: Simscape Component

Ideal Capacitor

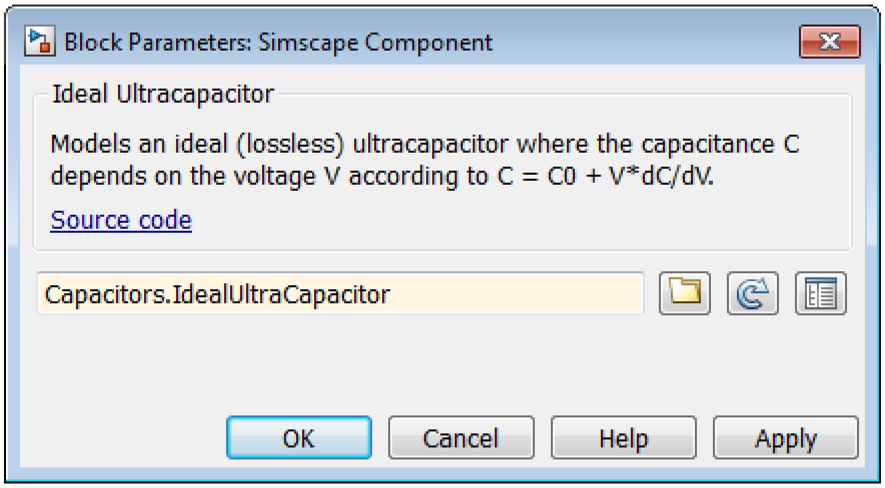
Models an ideal (lossless) capacitor. The output current I is related to the input voltage V by I = C*dV/dt where C is the capacitance.

Source code

Capacitors.IdealCapacitor

10 Click 📁. The browser opens in the `+Capacitors` directory, because it contains the currently selected component.

11 Select the `IdealUltraCapacitor.ssc` file and click **Open**. The name of the source file appears in the text field of the source file selector dialog box, and the block name, description, and link to source code appear in the preview pane.


Block Parameters: Simscape Component

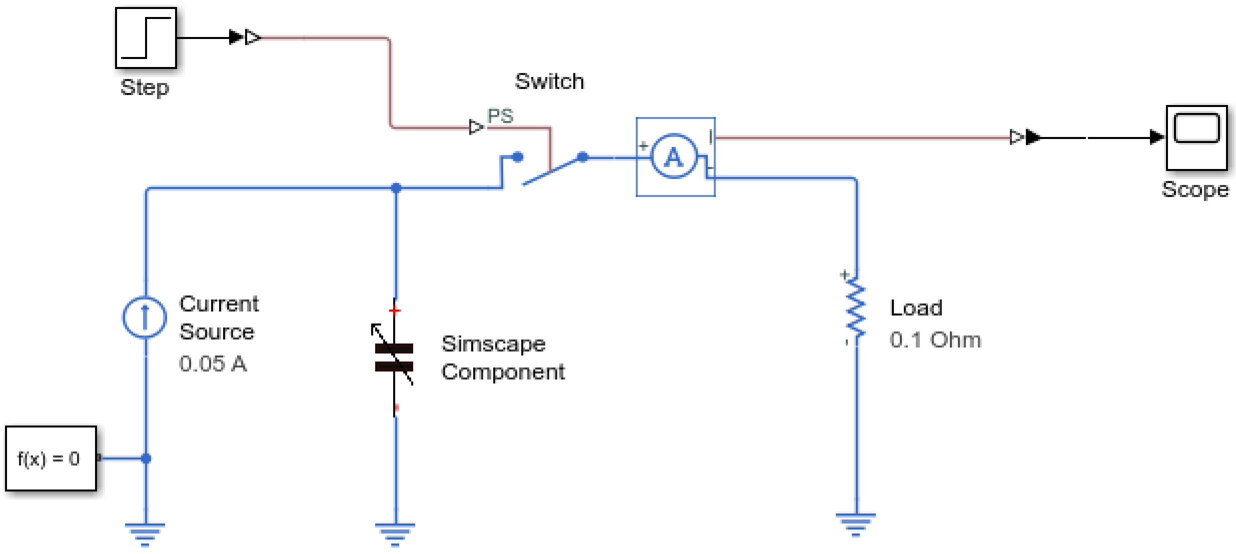
Ideal Ultracapacitor

Models an ideal (lossless) ultracapacitor where the capacitance C depends on the voltage V according to C = C0 + V*dC/dV.

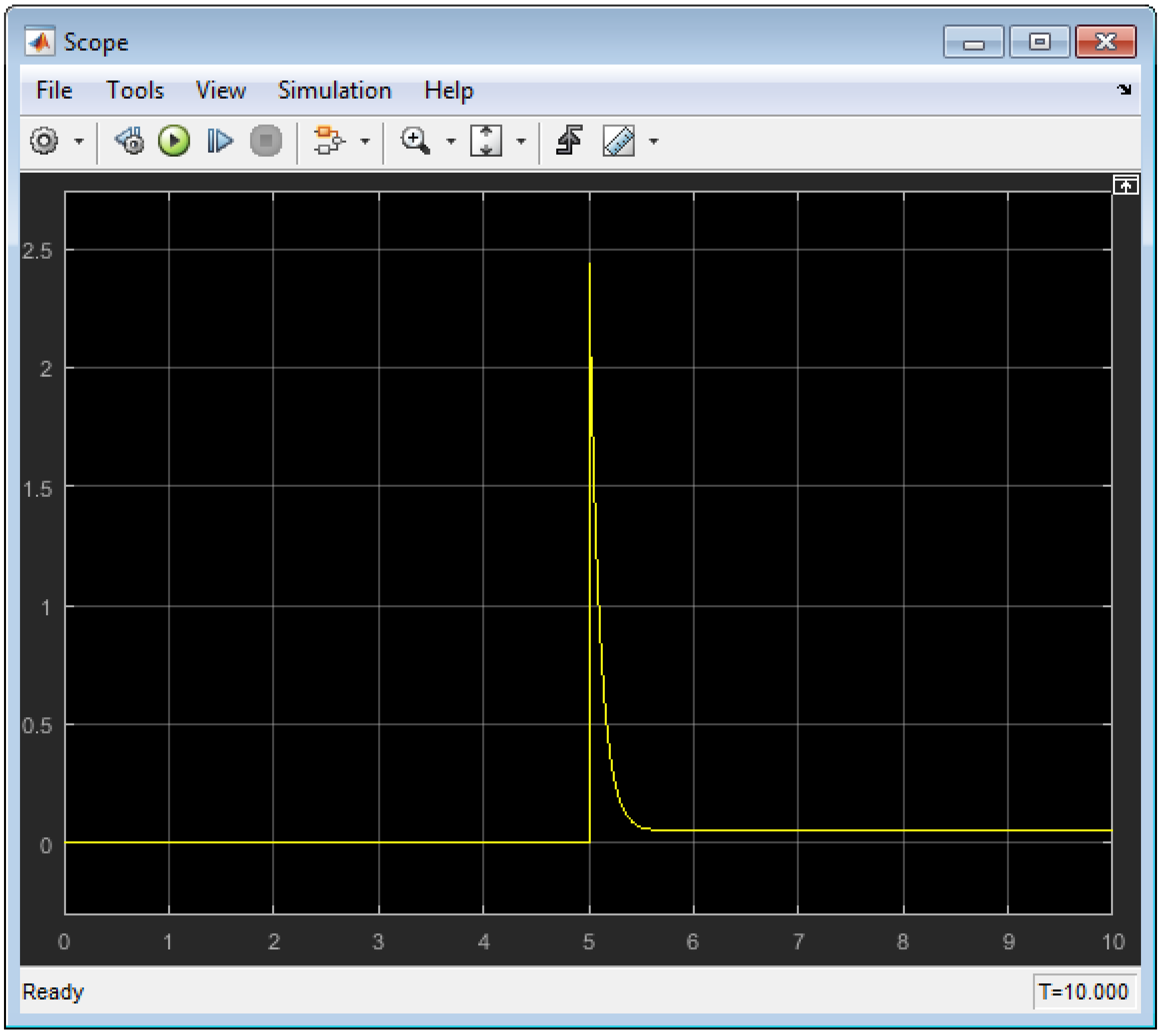
Source code

Capacitors.IdealUltraCapacitor

12 Click **OK**. The block icon in the model diagram updates to reflect the new source component.

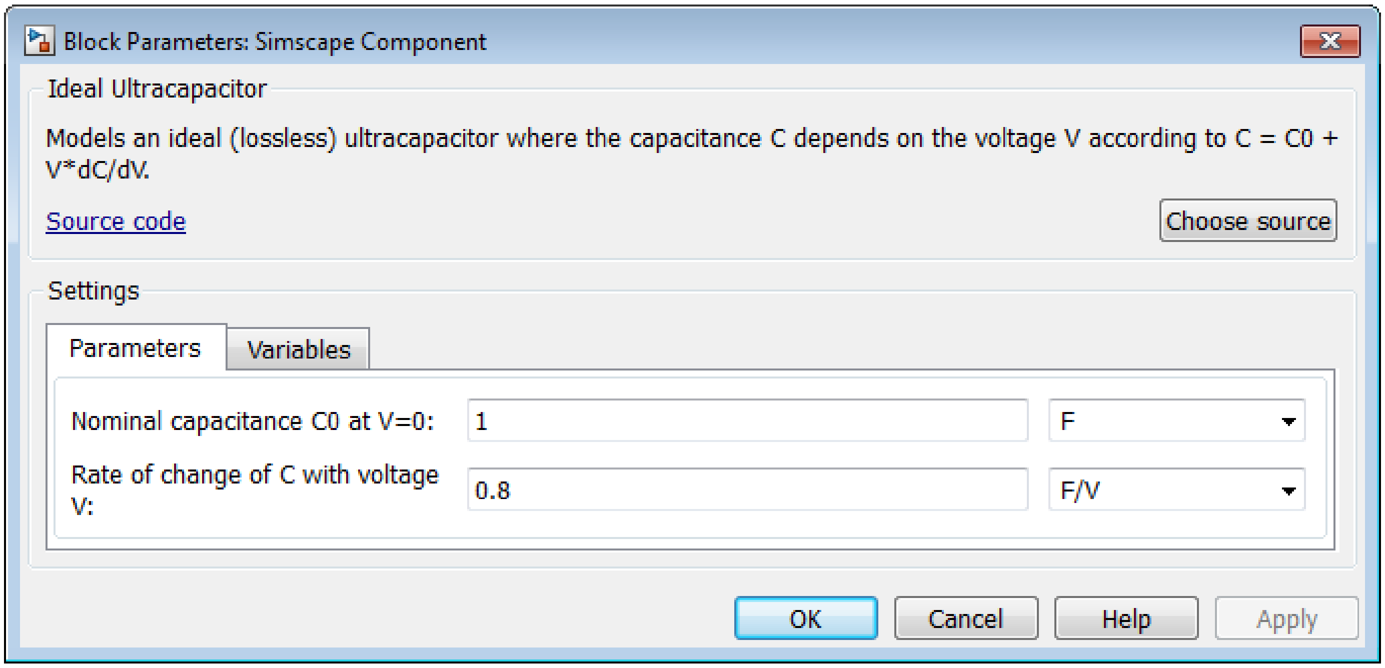


13 Rerun the simulation.



Simulation results show that, when the switch is flipped at t=5 seconds, the current delivered to the load is less than 2.5 A.

14 To make the effect more pronounced, open the block dialog box and increase the **Rate of change of C with voltage V** parameter value to 0.8 F/V.


Block Parameters: Simscape Component

Ideal Ultracapacitor

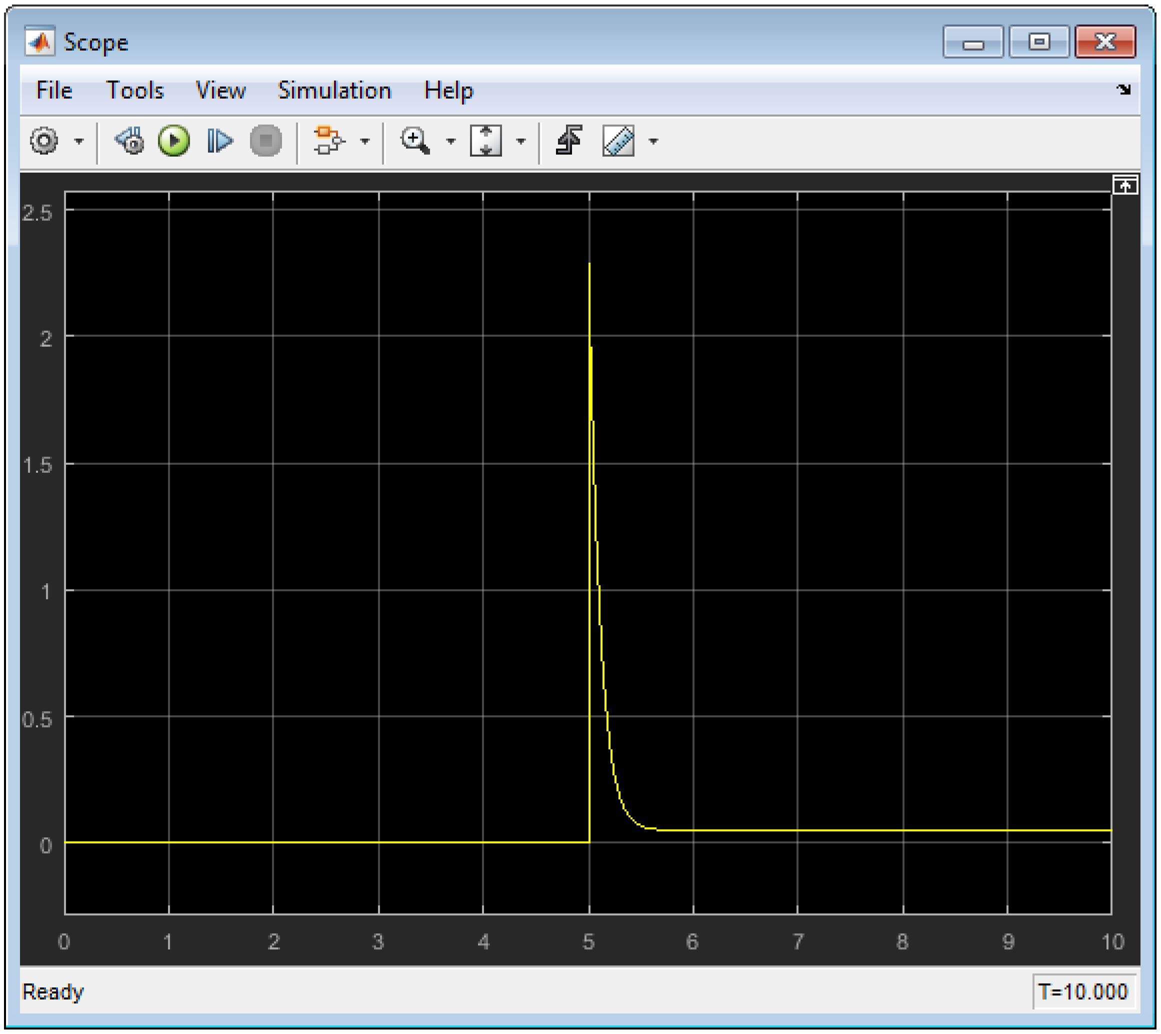
Models an ideal (lossless) ultracapacitor where the capacitance C depends on the voltage V according to C = C0 + V*dC/dV.

Source code

Settings

Parameters | Variables

Nominal capacitance C0 at V=0:	1	F
Rate of change of C with voltage V:	0.8	F/V



See Also

Related Examples

- "Deploy a Component File in Block Diagram" on page 4-5
- "Prototype a Component and Get Instant Feedback" on page 4-18

More About

- "Selecting Component File Directly from Block" on page 4-3
- "Customizing the Block Name and Appearance" on page 4-33

Prototype a Component and Get Instant Feedback

This example shows how you can interactively modify the component source and get instant feedback on the resulting block implementation.

To have the block reflect the changes to the underlying source, right-click the block icon and, from the context menu, select **Simscape** > **Refresh source code**. If you make a mistake (for example, omit the `end` keyword) when editing the component source, then when you refresh the block, the compiler issues a diagnostic error message, pointing to the appropriate line in the code.

1 Open the Simscape > Foundation Library > Electrical > Electrical Elements > Variable Resistor block dialog box and click the **Source code** link. The underlying source code opens in the Editor window.

```
component variable_resistor
% Variable Resistor :1.5
% 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. The Minimum resistance parameter prevents
% negative resistance values.
%
% Connections + and - are conserving electrical ports corresponding to
% the positive and negative terminals of the resistor respectively. The
% current is positive if it flows from positive to negative, and the
% voltage across the resistor is given by V(+)-V(-).

% Copyright 2005-2019 The MathWorks, Inc.

inputs
    R = { 0.0, 'Ohm' }; % R:left
end

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

parameters
    Rmin = { 0, 'Ohm' }; % Minimum resistance R>=0
end

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

branches
    i : p.i -> n.i;
end

equations
    assert(Rmin>=0)
    v == p.v - n.v;
    if R > Rmin
        v == i*R;
    else
        v == i*Rmin;
    end
end

end
```

2 Change the component name in the first line:

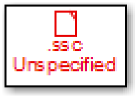
```
component my_var_res
```

3 Save the source code as a file called `my_var_res.ssc` in your current working directory.

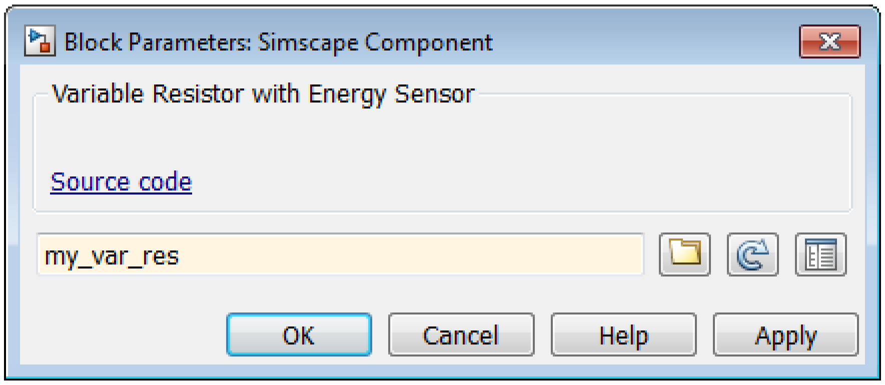
4 To create a new model with optimal settings for physical modeling, in the MATLAB Command Window, type:

```
ssc_new
```

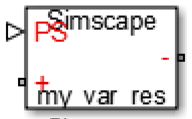
5 Open the Simscape > Utilities library and add the Simscape Component block to your model. At first, the block does not point to any component file, therefore it does not have any ports and the block icon says `Unspecified`.



6 Double-click the block to open the source file selector dialog box. Type `my_var_res` into the text field.

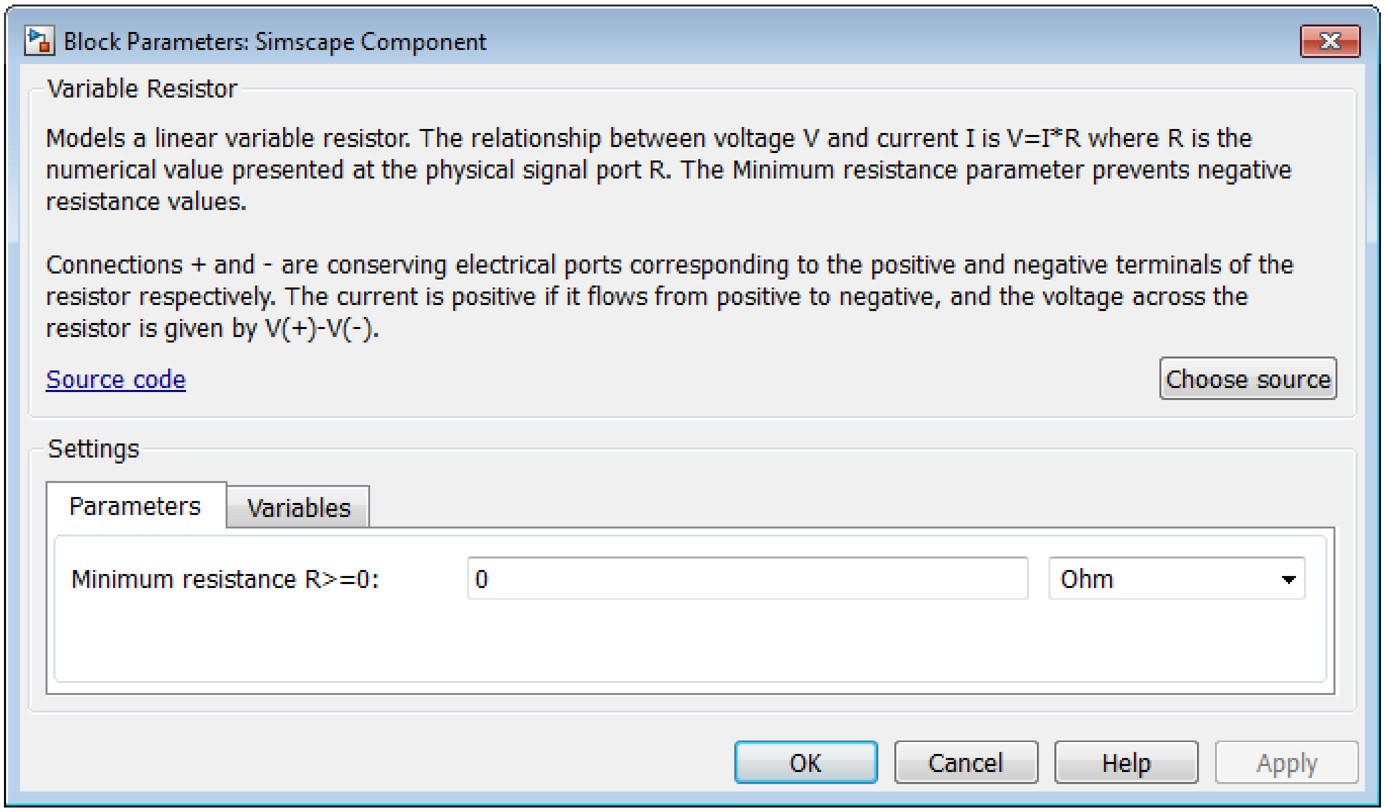

Block Parameters: Simscape Component
Variable Resistor with Energy Sensor
Source code
my_var_res
OK
Cancel
Help
Apply

7 Click **OK**. The block icon gets updated, reflecting the selected source component. It now has two conserving electrical ports, + and –, and a physical signal input port PS.



Simscape Component

8 Double-click the block to open its dialog box. At this point, it has the same block name, description, parameters, and variables, as the Variable Resistor block in the Foundation library.



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. The Minimum resistance parameter prevents negative resistance values.

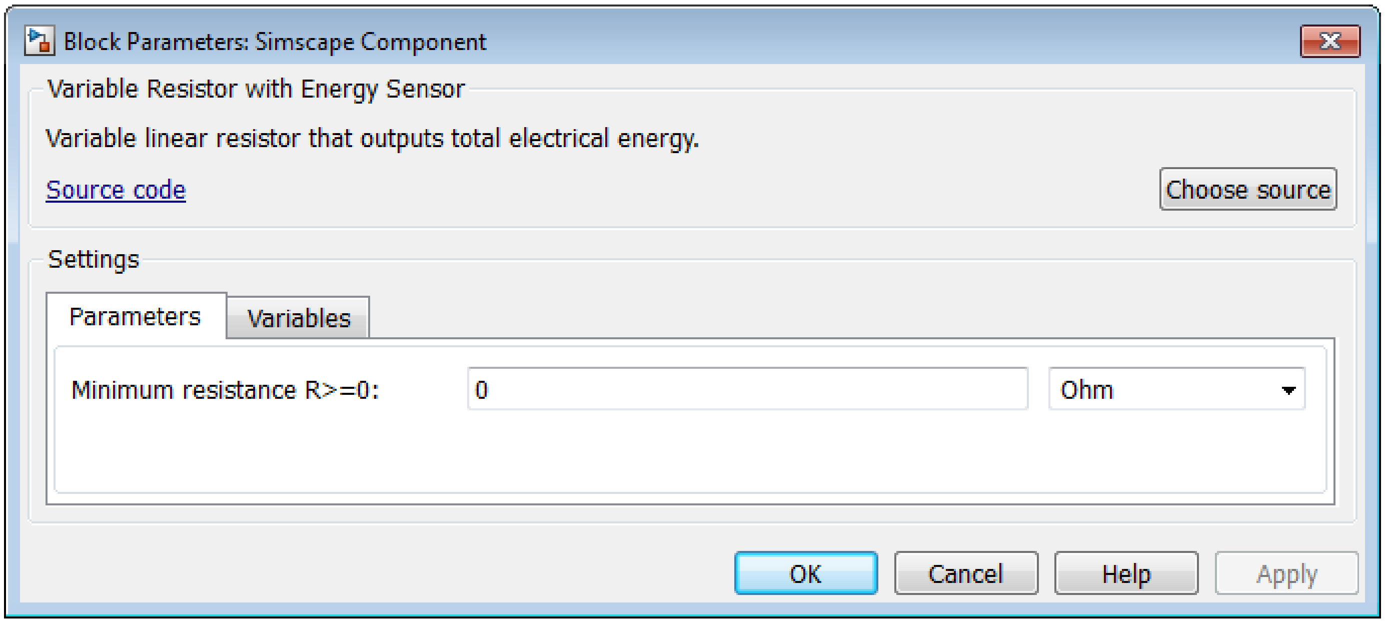
Connections + and - are conserving electrical ports corresponding to the positive and negative terminals of the resistor respectively. The current is positive if it flows from positive to negative, and the voltage across the resistor is given by V(+)-V(-).

Minimum resistance R>=0: 0 Ohm

9 Click the **Source code** link to start editing the source code. Change the block name and description:

```
component my_var_res
% Variable Resistor with Energy Sensor
% Variable linear resistor that outputs total electrical energy.
```

10 To have the block reflect the changes to the underlying source, right-click the block icon and, from the context menu, select **Simscape** > **Refresh source code**. The block dialog box updates accordingly.



Variable Resistor with Energy Sensor

Variable linear resistor that outputs total electrical energy.

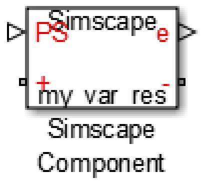
Minimum resistance R>=0: 0 Ohm

11 Declare the output e and add the equation calculating total electrical energy. The component source now looks like this:

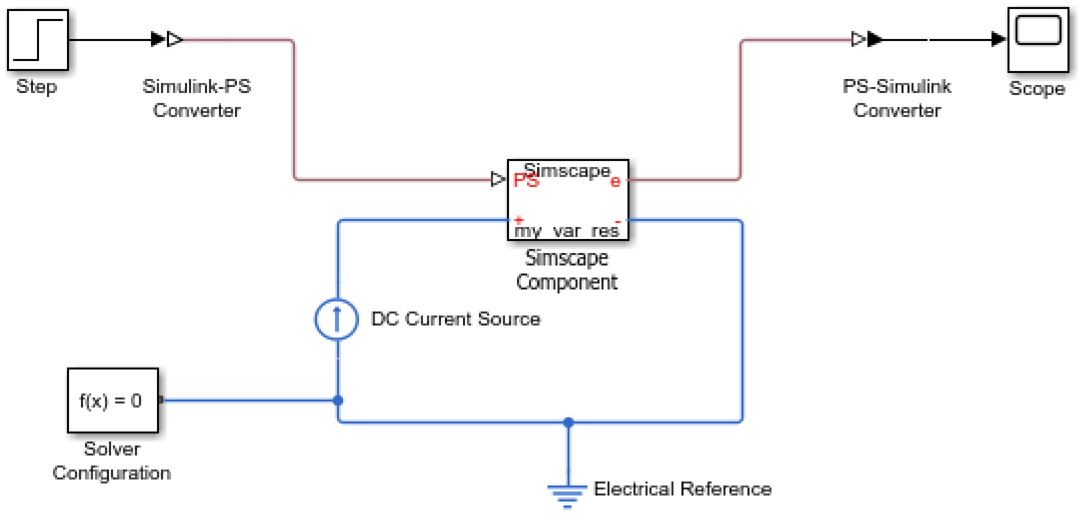
```
component my_var_res
% Variable Resistor with Energy Sensor
% Variable linear resistor that outputs total electrical energy.

inputs
    R = { 0.0, 'Ohm' }; % PS:left
end

outputs
    e = { 0, 'J' };
end

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

parameters
    Rmin = { 0, 'Ohm' }; % Minimum resistance R>=0
end

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

branches
    i : p.i -> n.i;
end

equations
    assert(Rmin>=0)
    v == p.v - n.v;
    if R > Rmin
        v == i*R;
    else
        v == i*Rmin;
    end
    e == integ(v*i);
end
```

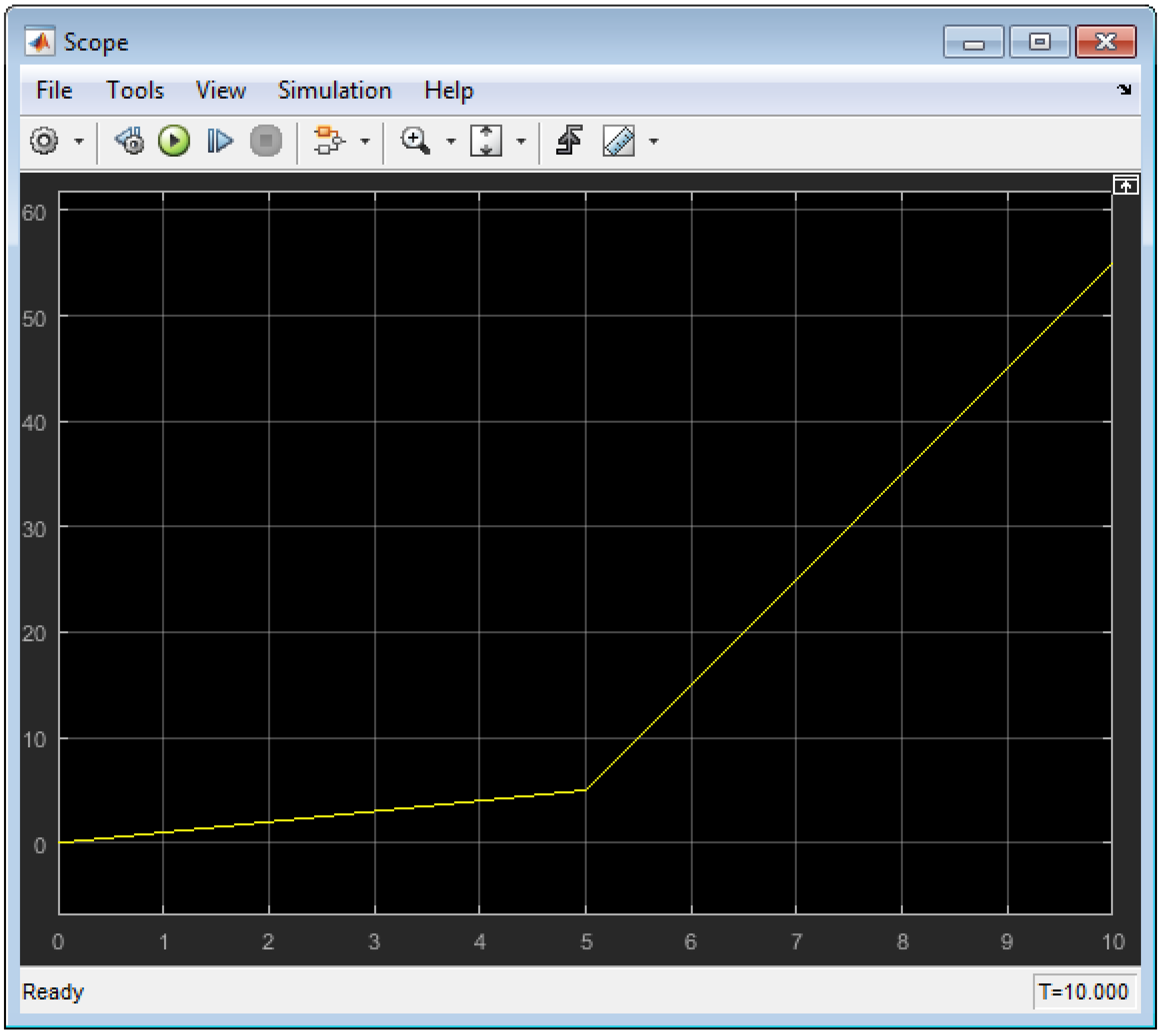
```
end
```

12 Refresh the block again. The block icon now has an additional physical signal output port e.



13 Connect the block to a simple test rig to verify the correct performance.





Note There is a limitation that the name of the model cannot be the same as the name of the source file for the Simscape Component block. Therefore, if you save the test rig model, make sure to give it a different name, such as `my_var_res_test`.

See Also

Related Examples

- “Model Linear Resistor in Simscape Language” on page 1-3
- “Customize Block Display” on page 4-45
- “Deploy a Component File in Block Diagram” on page 4-5
- “Switch Between Different Source Components” on page 4-9

More About

- “Selecting Component File Directly from Block” on page 4-3

- “Customizing the Block Name and Appearance” on page 4-33

Building Custom Block Libraries

In this section...

Workflow Overview

To generate a custom block library from Simscape component files, follow these steps:

1. Organize your Simscape files on page 4-25. Simscape files must be saved in package directories. The package hierarchy determines the resulting library structure.
2. Optionally, provide source protection on page 4-26. 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. Build the custom block library on page 4-26. 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. 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. By default, the custom block library generated from this package will be called `SimscapeCustomBlocks_lib` (you can specify a different name). The library will have three sublibraries with names corresponding to the package subdirectories (`Electrical`, `Hydraulic`, and `Mechanical`). For information on building custom block libraries, see "Converting Your Simscape Files" on page 4-26.

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 4-25. For information on building custom block libraries, see “Converting Your Simscape Files” on page 4-26.

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 on page 4-26.

For example, you may have a top-level package directory, where you store your Simscape files, named `+SimscapeCustomBlocks`. You can generate a custom block library either from the package parent directory, or from a directory inside the package. From the package parent directory, 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.

When building a custom library from a package, `ssc_build` lets you specify a different name and location for the library file than the default ones. For more information, see `ssc_build`.

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.

See Also

`ssc_build` | `ssc_mirror` | `ssc_protect`

Related Examples

- "Create a Custom Block Library" on page 4-31

More About

- "Customizing the Library Name and Appearance" on page 4-29
- "When to Rebuild a Custom Library" on page 4-28

When to Rebuild a 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 run `ssc_clean` and then 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 4-26.

Customizing the Library Name and Appearance

In this section…

Library Configuration Files

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 that you want to display in the sublibrary.	No annotation in the library	Used in annotation for 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 4-30, 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

Option	Usage	Description	Default	For Top-Level Package
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 “Library Configuration Files” on page 4-29.

See Also

Related Examples

- “Create a Custom Block Library” on page 4-31

More About

- “Building Custom Block Libraries” on page 4-25

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 4-25, “Converting Your Simscape Files” on page 4-26, and “Customizing the Library Name and Appearance” on page 4-29.

Consider the following directory structure:

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

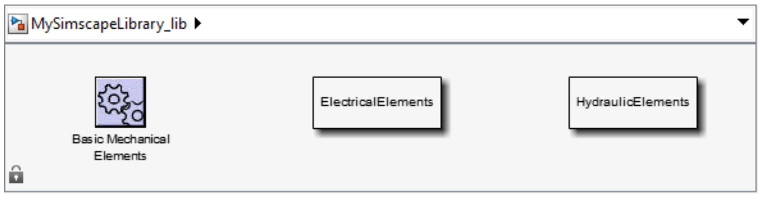
This means that you have a top-level package called `+MySimscapeLibrary`, which contains three subpackages, `+MechanicalElements`, `+ElectricalElements`, and `+HydraulicElements`. The `+MechanicalElements` package contains two component files, `inertia.ssc` and `spring.ssc`, a library icon file `lib.jpg`,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

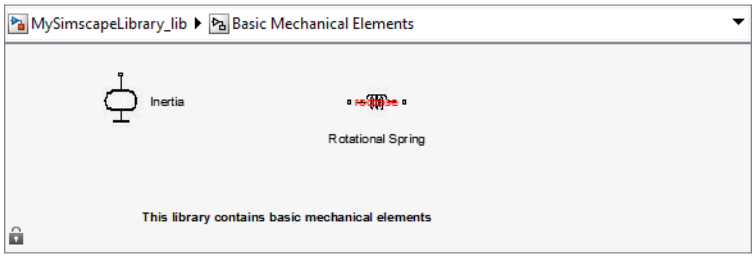
```
ssc_build MySimscapeLibrary;
```

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:



This library contains basic mechanical elements

Customizing the Block Name and Appearance

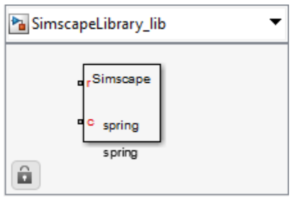
In this section...

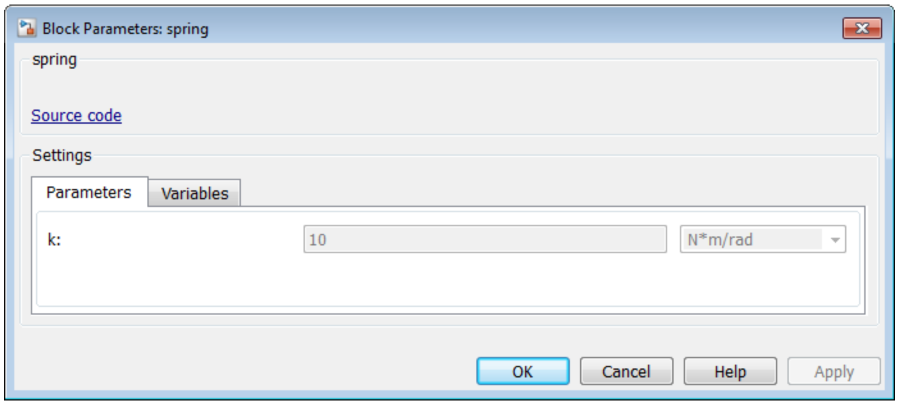
Default Block Display

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

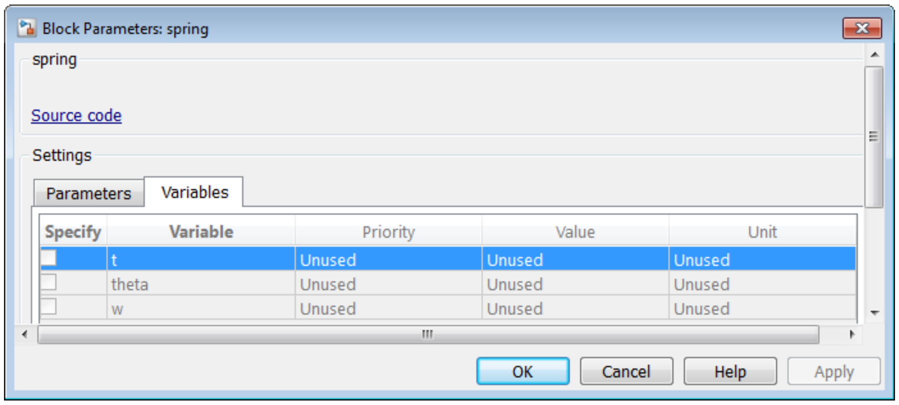
The following example shows a component file, named `spring.ssc`, and the resulting library block and dialog box.

```
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
  branches
    t : r.t -> c.t;
  end
  equations
    assert(k>0)
    w == r.w - c.w;
    t == k * theta;
    w == theta.der;
  end
end
```





k: 10 N*m/rad



Specify	Variable	Priority	Value	Unit
☐	t	Unused	Unused	Unused
☐	theta	Unused	Unused	Unused
☐	w	Unused	Unused	Unused

If you click the **Source code** 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 for its parameters and variables 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.

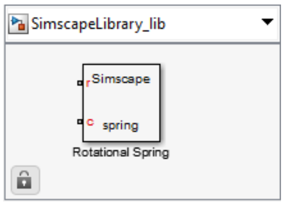
Customize the Block Name

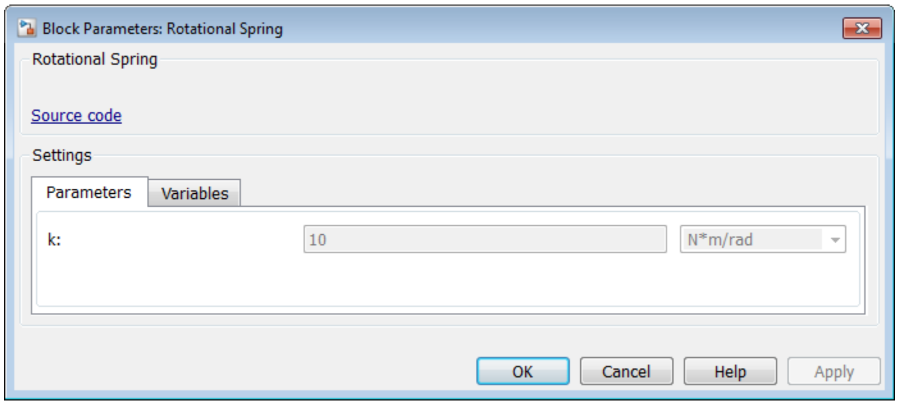
To provide a more descriptive name for the block than the name of the component file, put it on a separate comment line just below the `component` declaration. The comment line must begin with the `%` character. The entire content of this line, following the `%` character, is interpreted as the block 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:





Rotational Spring

Source code

Settings

Parameters | Variables

k: 10 N*m/rad

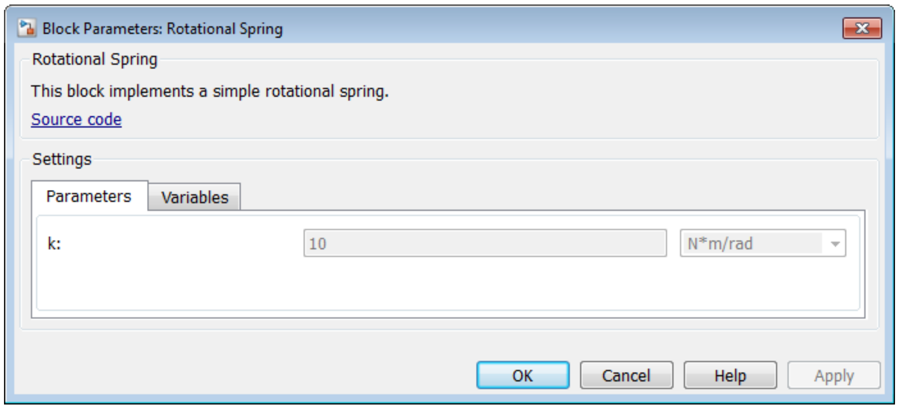
Describe the Block Purpose

The previous section on page 4-34 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:



Rotational Spring

This block implements a simple rotational spring.

Source code

Settings

Parameters | Variables

k: 10 N*m/rad

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

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

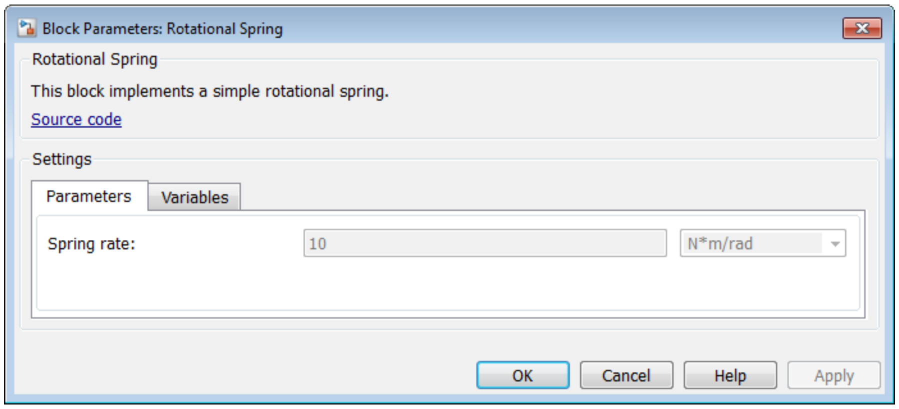
Specify Meaningful Names for the Block Parameters and Variables

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:


Block Parameters: Rotational Spring

Rotational Spring

This block implements a simple rotational spring.

Source code

Settings

Parameters | Variables

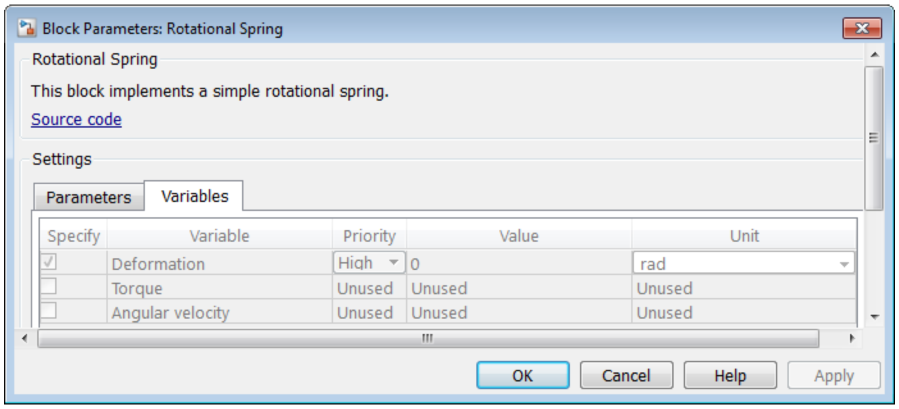
Spring rate: 10 N*m/rad

Use the same technique to specify meaningful names for the top-level public variables of the component. These variables appear on the **Variables** tab of the block dialog box, and giving them descriptive names helps with the block-level variable initialization prior to simulation.

For example, if you have the following component file:

```
component spring
%Rotational Spring
% This block implements a simple rotational spring.
...
  variables
    theta = { value = { 0 , 'rad' }, priority = priority.high }; % Deformation
    t = { 0, 'N*m' };   % Torque
    w = { 0, 'rad/s' }; % Angular velocity
  end
...
end
```

the resulting **Variables** tab of the block dialog box looks like this:



Block Parameters: Rotational Spring

Rotational Spring

This block implements a simple rotational spring.

Source code

Settings

Specify	Variable	Priority	Value	Unit
✓	Deformation	High	0	rad
☐	Torque	Unused	Unused	Unused
☐	Angular velocity	Unused	Unused	Unused

Group and Reorder Block Parameters Using Annotation

By default, custom block generated from a component has all the component parameters with `ExternalAccess=modify` listed in declaration order in a single tab, titled **Parameters**. If there are variables with `ExternalAccess=modify`, the block dialog also contains a separate **Variables** tab. Similarly, in the Property Inspector, parameters and variables are listed in declaration order in two separate tree nodes, **Parameters** and **Variables**.

For complex components with large numbers of parameters, you can enhance the block usability by grouping parameters based on their function or on the effect that they model. For example, you can separate electrical and mechanical parameters for a motor, or place all temperature-dependent parameters in a separate group. You do this by using the block layout annotation, `UILayout`.

`UILayout` annotation lets you define titled groups of component parameters, the order of these groups, and the order of parameters in each group. When you deploy the component as a custom Simscape block, these groups translate into dialog box tabs (and into Property Inspector tree nodes).

The block layout annotation syntax is:

```
annotations
  UILayout = [UIGroup("Title 1",p1,p2)
              UIGroup("Title 2",p3)]
end
```

The following rules apply:

- `UILayout` is a class-level annotation, meaning that it can appear only once per component file.
- `UILayout` annotation must contain a nonempty array of `UIGroup` constructs. The order of the groups defines the order of the dialog box tabs.
- Each `UIGroup` construct must include a title string and a nonempty comma-separated list of component parameters. The title string serves as the title of the dialog box tab, and the listed parameters appear on that tab, in list order.
- Component parameters with `ExternalAccess=modify` not listed in any of the groups appear at the end of the first tab in declaration order.
- A parameter cannot belong to more than one group. Listing the same parameter in multiple groups results in an error.
- This annotation does not affect component variables. Whether you use the block layout annotation or not, variables are listed on a separate **Variables** tab in declaration order.

Use the block layout annotation to:

- Create multiple tabs in a block dialog box. To do this, define multiple groups of parameters. Each group is displayed on a separate tab.
- Change the parameter order in the block dialog box, compared to the declaration order. To do this, define one group of parameters, titled `"Parameters"`, and list the component parameters in desired order.

This feature makes authoring the component source independent of the resulting block layout. You can arrange parameter declaration blocks in a way that helps code readability, and later reorder parameters in the block interface, as needed. For a detailed example, see “Use Advanced Techniques to Customize Block Display” on page 4-47.

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.

There are two ways to control the port label and location in the block icon:

- Use separate controls for port labels and locations. This is the recommended way because it provides more flexibility. Specify the port labels by using comments immediately following the node, input, or output declaration, similar to specifying meaningful names for parameters and variables. Specify the port side separately, by using the `annotations` section in the component file. For more information, see “Customize Port Labels on the Block Icon” on page 4-38 and “Control Port Locations Using Annotations” on page 4-39.
- Use comments immediately following the node, input, or output declaration, to specify both the name and location of the block port. This is a legacy method that allows you to have ports only on two opposite sides of the block icon (left and right, or top and bottom). For more information, see “Use Comments to Control Port Locations” on page 4-40.

Customize Port Labels on the Block Icon

The default port label corresponds to the name of the node, input, or output, as specified in the declaration block. Similar to specifying meaningful names for block parameters and variables, you can customize a port label by supplying a comment immediately following the node, input, or output declaration. For example:

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

If you specify an empty comment string after a node, input, or output declaration, the corresponding port will not have its name displayed. For example, the following syntax suppresses the port label for the physical signal input port `PS`:

```
inputs
   PS; %
end
```

Control Port Locations Using Annotations

Use the `annotations` section in the component file to specify the port locations. For example:

```
nodes
  H = foundation.thermal.thermal;
  p = foundation.electrical.electrical; % +
  n = foundation.electrical.electrical; % -
end
annotations
  H : Side = top;
  p : Side = left;
  n : Side = right;
end
```

Rules and restrictions:

- You can use `Side` annotations for nodes, inputs, and outputs.
- Member attributes must be uniquely defined. Therefore, you cannot use the same member ID on the left side of the `Side` annotations more than once.

  ```
  nodes
      p = foundation.electrical.electrical;
      n = foundation.electrical.electrical;
  end
  annotations
       [p, n] : Side = left;
       n : Side = right;   % error: multiple definitions for 'Side' of port 'n'
  end
  ```

  If you specify different locations for the same port by using both the annotations and the comments, the location in the `annotations` section takes precedence.

  ```
  outputs
    o = {0, 'V'}; %o:right
  end
  annotations
     o : Side = top;  % annotation takes precedence, so port will be located on the top
  end
  ```

- You can use the same member ID multiple times to declare different annotation attributes.

  ```
  nodes
      n1 = foundation.electrical.electrical;
      n2 = foundation.electrical.electrical;
      n3 = foundation.electrical.electrical;
  end
  annotations
      [n1, n2] : ExternalAccess = none;
      [n2, n3] : Side = right;
  end
  ```

- Similarly, you can declare different annotation attributes for the same member ID in one statement.

  ```
  nodes
      n1 = foundation.electrical.electrical;
      n2 = foundation.electrical.electrical;
      n3 = foundation.electrical.electrical;
  end
  annotations
      [n1, n2] : ExternalAccess = none, Side = top;
      n3 : Side = right;
  end
  ```

- You cannot conditionally switch port sides, that is, include `Side` annotations in branches of a conditional statement.

  ```
  parameters
      thermal_effects = false; % Model thermal effects?
  end
  nodes (ExternalAccess=none)
     H = foundation.thermal.thermal;
  end
  if thermal_effects
    % Expose thermal port
      annotations
         H : ExternalAccess = modify;
         H : Side = bottom; % error: cannot have 'Side' annotations inside conditional sections
      end
  end
  ```

  To place a conditionally visible port on a specific side of the block (for example, on the bottom), use the following syntax:

  ```
  parameters
      thermal_effects = false; % Model thermal effects?
  end
  nodes (ExternalAccess=none)
     H = foundation.thermal.thermal;
  end
  annotations
     H : Side = bottom;
  end
  if thermal_effects
    % Expose thermal port
      annotations
         H : ExternalAccess = modify;
      end
  end
  ```

Use Comments to Control Port Locations

Note This is a legacy method that has multiple limitations. Therefore, the recommended method is to use the `annotations` section, as described in “Control Port Locations Using Annotations” on page 4-39.

You can also use a comment that immediately follows the node, input, or output declaration to specify both the port label and location. The comment 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).

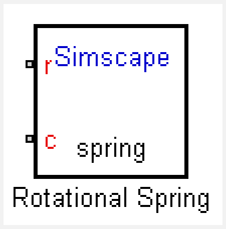
You can also leave the port label field empty and specify just the location. In this case, the port will not have its name displayed. For example, the following syntax suppresses the port label and locates it on the top of the block icon:

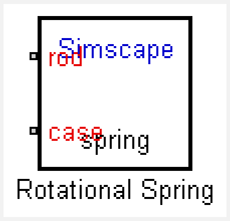
```
r = foundation.mechanical.rotational.rotational; % :top
```

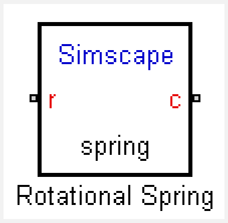
If you specify an empty comment string after a node, input, or output declaration, the corresponding port will not be labeled and will be located on the left side of the block icon.

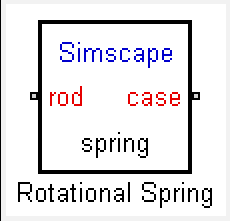
The following are examples of node declarations and the resulting block icons.

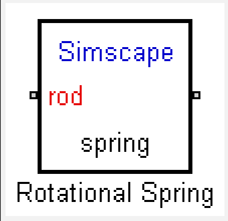
Syntax	Block Icon

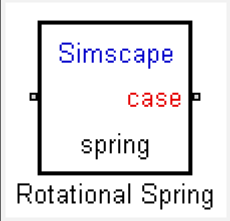
`nodes`
`    r = foundation.mechanical.rotational.rotational;`
`    c = foundation.mechanical.rotational.rotational;`
`end`
	

`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`
	

`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

The default block icon is a rectangle displaying the block name. You can replace this default icon with a custom image file. For information on supported file formats and image properties, see “Supported File Formats” on page 4-43.

There are two ways to specify a custom block icon:

- Explicitly, using the `annotations` section in the component file. This is the recommended way because it provides more flexibility. You can keep the image files in a separate folder and specify relative paths for the block icons. You can also specify conditional custom icons for different block variants. For more information, see “Using Annotations” on page 4-42.
- Implicitly, using the file naming conventions. This method is convenient if you ship complete library packages to customers. For more information, see “Using File Naming Conventions” on page 4-42.

Using Annotations

Use the `annotations` section in the component file to specify the name of the custom block icon. The file name must contain the file extension. For example:

```
  annotations
    Icon = 'custom_spring.jpg';
  end
```

The file name can include a relative path from the folder containing the component file to the folder containing the image file, for example:

```
  annotations
    Icon = '../../block_icons/custom_spring.jpg';
  end
```

The `annotations` section also lets you specify conditional custom icons. This is especially useful if the number of ports changes for different variants. For example:

```
component MyPipe
  parameters
    thermal_variant = false; % Model thermal effects?
  end
  if thermal_variant
  % Use icon with additional thermal port
    annotations
       Icon = 'pipe_thermal.jpg';
    end
  else
  % Use regular icon, with two fluid ports
    annotations
       Icon = 'pipe.jpg';
    end
  end
  [...] % Other parameters, variables, nodes, branches, equations
end
```

Using File Naming Conventions

Instead of explicitly specifying a custom block icon using the `annotations` section, you can do it implicitly, by placing an image file with the same name as the component in the folder containing the component file.

This method is convenient if you ship complete library packages to customers. For example, if the subpackage containing the component file `spring.ssc` also contains a file named `spring.jpg`, then that image file is automatically used for the icon representing this block in the custom library.

The implicit rules for using custom block icons are:

1. If the `annotations` section does not explicitly specify a custom icon image, or if that image is not found, the software looks in the folder containing the component file for an image file with the same name as the component.
2. If there are multiple image files with the same name, the formats take precedence in the order listed in "Supported File Formats" on page 4-43. For example, if the subpackage contains both `spring.jpg` and `spring.bmp`, `spring.jpg` is the image that will appear in the custom library.

Supported File Formats

The following image file formats are supported for custom block icons:

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

Caution Using `svg` format together with domain-specific line styles can lead to unexpected results, because domain line styles and colors can propagate to parts of the custom block icon. For more information on turning domain-specific line styles on and off, see "Domain-Specific Line Styles".

The image type must be an RGB (truecolor) image, stored as an *m*-by-*n*-by-3 data array. For more information, see "RGB (Truecolor) Images".

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 4-34. 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.

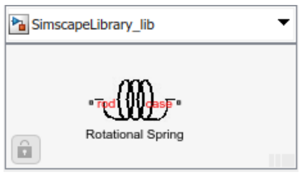
See Also
`annotations`

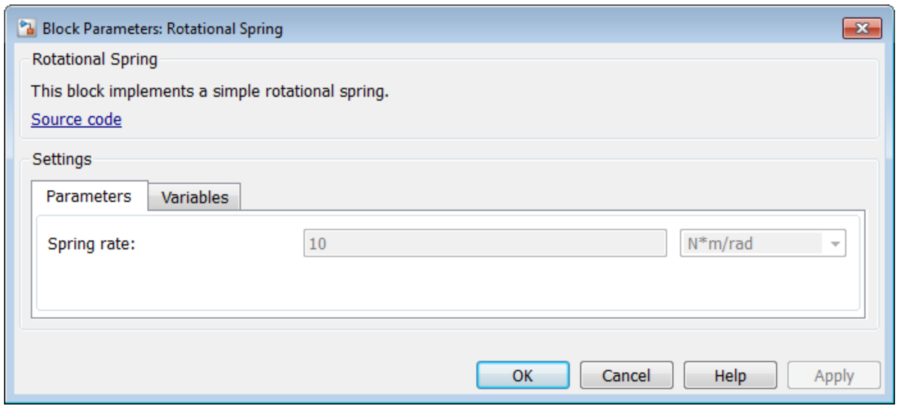
Related Examples
- “Customize Block Display” on page 4-45
- “Use Advanced Techniques to Customize Block Display” on page 4-47

Customize Block Display

This example shows a complete component file with annotation and the resulting library block and dialog box. The image file, `custom_spring.jpg`, is located in the same folder as the component file. This example is an illustration of basic techniques described in “Customizing the Block Name and Appearance” on page 4-33.

```
component spring
% Rotational Spring
% This block implements a simple rotational spring.
  nodes
    r = foundation.mechanical.rotational.rotational; % rod
    c = foundation.mechanical.rotational.rotational; % case
  end
  annotations
    r : Side = left;
    c : Side = right;
    Icon = 'custom_spring.jpg';
  end
  parameters
    k = { 10, 'N*m/rad' }; % Spring rate
  end
  variables
    theta = { 0, 'rad' };  % Deformation
    t = { 0, 'N*m' };      % Torque
    w = { 0, 'rad/s' };    % Angular velocity
  end
  branches
    t : r.t -> c.t;
  end
  equations
    assert(k>0)
    w == r.w - c.w;
    t == k * theta;
    w == theta.der;
  end
end
```




Block Parameters: Rotational Spring

Rotational Spring

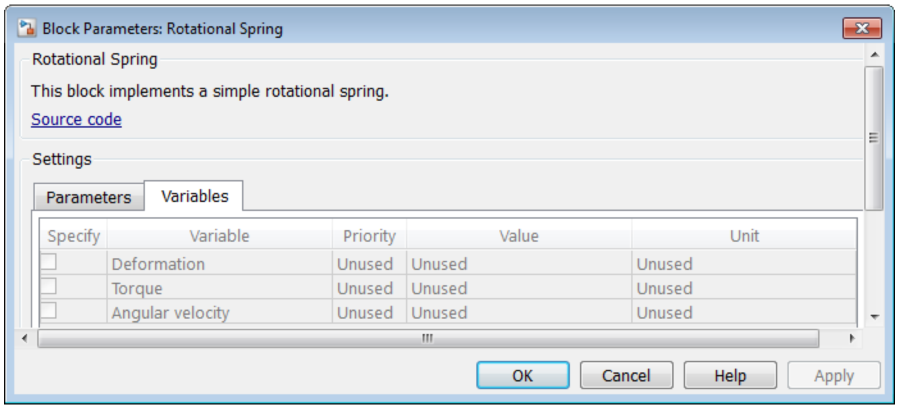
This block implements a simple rotational spring.

Source code

Settings

Parameters | Variables

Spring rate: 10 N*m/rad


Block Parameters: Rotational Spring

Rotational Spring

This block implements a simple rotational spring.

Source code

Settings

Parameters | Variables

Specify	Variable	Priority	Value	Unit
☐	Deformation	Unused	Unused	Unused
☐	Torque	Unused	Unused	Unused
☐	Angular velocity	Unused	Unused	Unused

See Also

`annotations`

More About

- "Customizing the Block Name and Appearance" on page 4-33
- "Use Advanced Techniques to Customize Block Display" on page 4-47

Use Advanced Techniques to Customize Block Display

This example shows how you can use block layout annotation and enumerations to improve usability of a custom block.

The following source code for the `DCMotorWithTabs` component shows declaration blocks for nodes and user-visible parameters, as well as control logic and annotations. The source code in this example does not include sections of code that have no effect on the block dialog box display, such as other declarations, intermediates, branches, and equations.

```
component DCMotorWithTabs
% DC Motor
% This block represents the electrical and torque characteristics of a
% DC motor.
%
% When a positive current flows from the electrical + to - ports, a
% positive torque acts from the mechanical C to R ports. Motor torque
% direction can be changed by altering the sign of the back-emf
% constant.

nodes
    p = foundation.electrical.electrical; % +:top
    n = foundation.electrical.electrical; % -:bottom
    R = foundation.mechanical.rotational.rotational; % R:top
    C = foundation.mechanical.rotational.rotational; % C:bottom
end

parameters
    Ra = {3.9, 'Ohm'};          % Armature resistance
    La = {12e-6, 'H'};          % Armature inductance
    Kv = {0.072e-3, 'V/rpm'};   % Back-emf constant
    J = {0.01, 'g*cm^2'};       % Rotor inertia
    lam = {0, 'N*m/(rad/s)'};   % Rotor damping
    speed0 = {0, 'rpm'};        % Initial rotor speed
    i_noload = {0, 'A'};        % No-load current
    V_i_noload = {1.5, 'V'};    % DC supply voltage when measuring no-load current
end

% Rotor damping control parameter
parameters
    r_damp = damping.direct;    % Rotor damping parameterization
end

% Conditional parameter visibility for Rotor damping parameterization
if r_damp == damping.direct
    annotations
        [i_noload,V_i_noload]: ExternalAccess=none;
    end
else
    annotations
        [lam]: ExternalAccess=none;
    end
end

annotations
    UILayout = [UIGroup("Electrical Torque",Ra,La,Kv,r_damp,i_noload,V_i_noload)
                UIGroup("Mechanical",J,lam,speed0)]
```

```
end

% Declarations with (ExternalAccess=none), branches, intermediates, equations

end
```

The `UILayout` annotation defines two groups, `Electrical Torque` and `Mechanical`, each with a list of parameters. When you generate a block from the `DCMotorWithTabs` component, each `UIGroup` becomes a tab in the block dialog box, the title string serves as the title of the tab, and the listed parameters appear on that tab, in list order.

In addition, the `DCMotorWithTabs` component provides two methods to specify rotor damping:

- Directly, using the **Rotor damping** parameter
- By specifying no-load current values instead, using two other parameters: **No-load current** and **DC supply voltage when measuring no-load current**

The `if` statement in the component source specifies the control logic for conditional parameter visibility, depending on the selected value of the control parameter, `r_damp` (**Rotor damping parameterization**). The control parameter uses an enumeration, located in a separate file, `damping.m`:

```
classdef damping < int32
   enumeration
     direct (0)
     derived (1)
   end
   methods(Static)
       function map = displayText()
         map = containers.Map;
         map('direct') = 'By damping value';
         map('derived') = 'By no-load current';
       end
   end
end
```

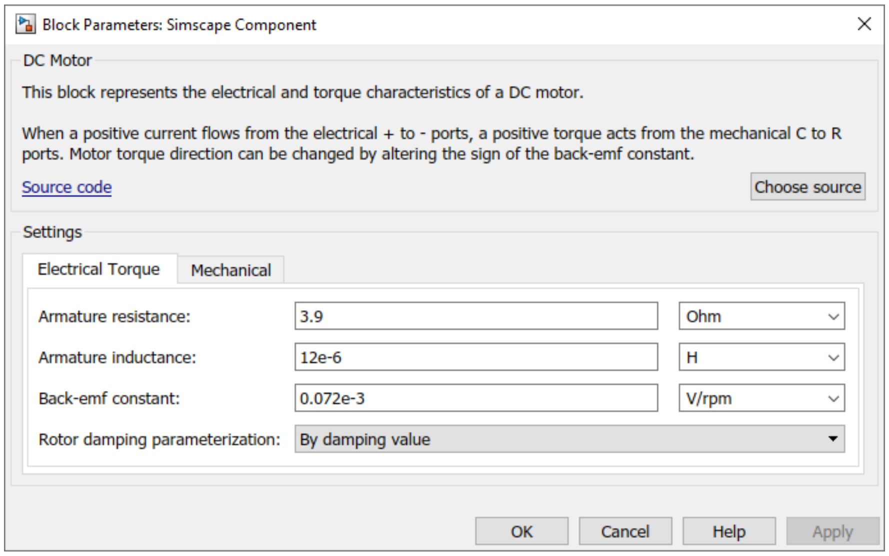
This enumeration file can be located either in same folder as the component file or on the MATLAB path. For more information, see “Specifying Display Strings for Enumeration Members” on page 3-16.

In the resulting block dialog, the **Rotor damping parameterization** parameter has a drop-down list of values:

- `By damping value`
- `By no-load current`

`By damping value` is the default value.

When you generate a block from the `DCMotorWithTabs` component, the block dialog box has two tabs:


Block Parameters: Simscape Component

DC Motor

This block represents the electrical and torque characteristics of a DC motor.

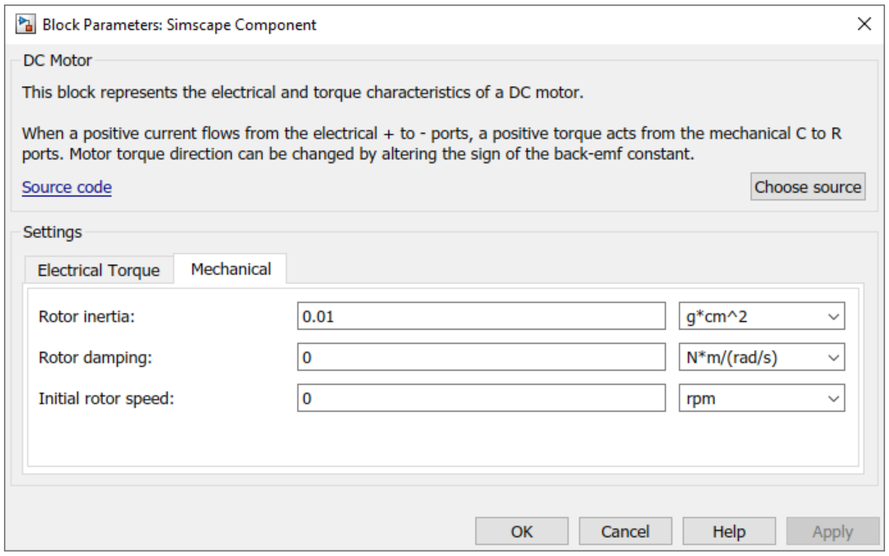
When a positive current flows from the electrical + to - ports, a positive torque acts from the mechanical C to R ports. Motor torque direction can be changed by altering the sign of the back-emf constant.

Source code

Settings

Electrical Torque | Mechanical

Armature resistance:	3.9	Ohm
Armature inductance:	12e-6	H
Back-emf constant:	0.072e-3	V/rpm
Rotor damping parameterization:	By damping value	


Block Parameters: Simscape Component

DC Motor

This block represents the electrical and torque characteristics of a DC motor.

When a positive current flows from the electrical + to - ports, a positive torque acts from the mechanical C to R ports. Motor torque direction can be changed by altering the sign of the back-emf constant.

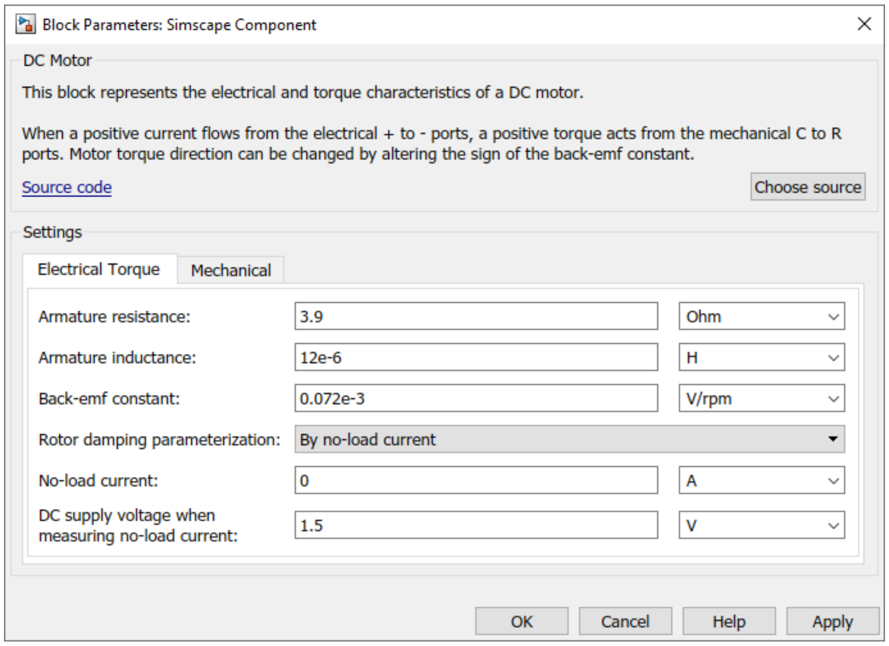
Source code

Settings

Electrical Torque | Mechanical

Rotor inertia:	0.01	g*cm^2
Rotor damping:	0	N*m/(rad/s)
Initial rotor speed:	0	rpm

If you set the **Rotor damping parameterization** parameter to `By no-load current`, two additional parameters appear on the **Electrical Torque** tab, and the **Rotor damping** parameter on the **Mechanical** tab is hidden.



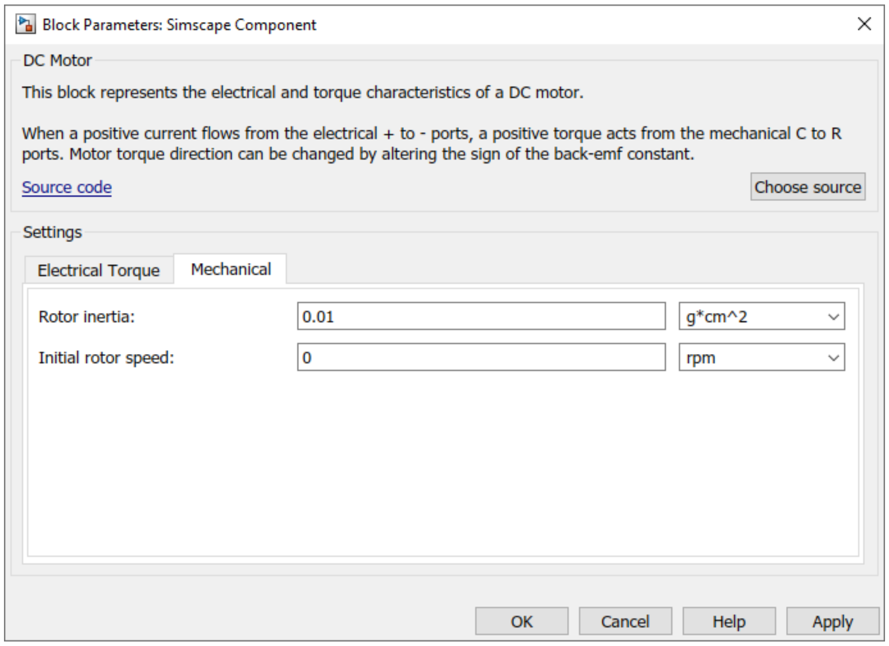
DC Motor

This block represents the electrical and torque characteristics of a DC motor.

When a positive current flows from the electrical + to - ports, a positive torque acts from the mechanical C to R ports. Motor torque direction can be changed by altering the sign of the back-emf constant.

Source code

Armature resistance:	3.9	Ohm
Armature inductance:	12e-6	H
Back-emf constant:	0.072e-3	V/rpm
Rotor damping parameterization:	By no-load current	
No-load current:	0	A
DC supply voltage when measuring no-load current:	1.5	V



DC Motor

This block represents the electrical and torque characteristics of a DC motor.

When a positive current flows from the electrical + to - ports, a positive torque acts from the mechanical C to R ports. Motor torque direction can be changed by altering the sign of the back-emf constant.

Source code

Rotor inertia:	0.01	g*cm^2
Initial rotor speed:	0	rpm

Note that in the **Electrical Torque** tab, the **No-load current** and **DC supply voltage when measuring no-load current** parameters appear below the **Rotor damping parameterization** parameter, even though they were declared earlier, in a separate declaration block. If the component was not using the block layout annotation, you could have achieved the same effect by changing the parameter declaration order, but that would have detracted from code readability.

See Also

`annotations`

More About

- "Customizing the Block Name and Appearance" on page 4-33
- "Defining Conditional Visibility of Component Members" on page 2-81
- "Enumerations" on page 3-15

Checking File and Model Dependencies

In this section...

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 Dependency Analyzer allows 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.

The Dependency Analyzer also includes dependencies for the Simscape blocks present in the model. For more information on the Dependency Analyzer, see "Analyze Model Dependencies".

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. On the **Modeling** tab of the Simulink Toolstrip, in the **Design** section, click **Dependency Analyzer**. 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 on page 6-4, 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 4-60.

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 "Building Custom Block Libraries" on page 4-25.

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.

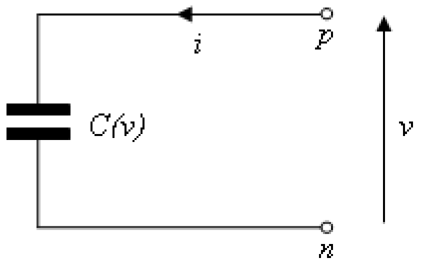
```
component IdealCapacitor
% Ideal Capacitor
% Models an ideal (lossless) capacitor. The output current I is related
% to the input voltage V by I = C*dV/dt where C is the capacitance.

% Copyright 2008-2017 The MathWorks, Inc.

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

    parameters
        C = { 1, 'F' }; % Capacitance
    end

    variables
        i = { 0, 'A'   }; % Current
        v = {value = { 0, 'V' }, priority = priority.high}; % Voltage drop
    end

    branches
        i : p.i -> n.i; % Through variable i from node p to node n
    end

    equations
        assert(C > 0)
        v == p.v-n.v; % Across variable v from p to n
        i == C*v.der; % Capacitor 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 4-33.

Also notice that in the equation section there is an assert 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:

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

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

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

and hence

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

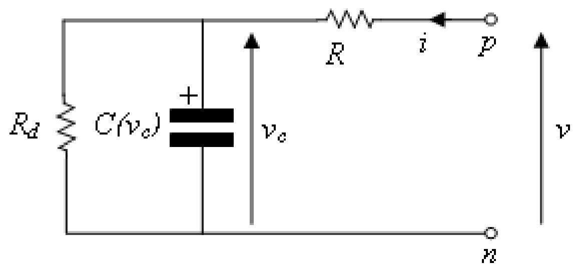
where $C_0 = 1$ and $C_v = 0.2$. This equation is implemented by the following line in the equation section of the Simscape file `IdealUltraCapacitor.ssc`:

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

In order for the Simscape software to interpret this equation, the variables (`v` and `i`) and the parameters (`C0` and `Cv`) must be defined in the declaration section. For more information, see “Declare Component Variables” on page 2-7 and “Declare Component Parameters” on page 2-11.

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 2008-2017 The MathWorks, Inc.

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

    parameters
        C0 = { 1, 'F' };    % Nominal capacitance C0 at V=0
        Cv = { 0.2, 'F/V'}; % Rate of change of C with voltage V
        R = {2, 'Ohm' };    % Effective series resistance
        Rd = {500, 'Ohm' }; % Self-discharge resistance
    end

    variables
        i = { 0, 'A'   }; % Current
        vc = {value = { 0, 'V' }, priority = priority.high}; % Capacitor voltage
    end

    branches
        i : p.i -> n.i; % Through variable i from node p to node n
    end

    equations
        assert(C0 > 0)
        assert(R > 0)
        assert(Rd > 0)
        let
            v = p.v-n.v; % Across variable v from p to n
        in
            i == (C0 + Cv*vc)*vc.der + vc/Rd; % Equation 1
            v == vc + i*R;                    % Equation 2
        end
    end

end
```

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

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

Summing voltages gives the second Simscape equation:

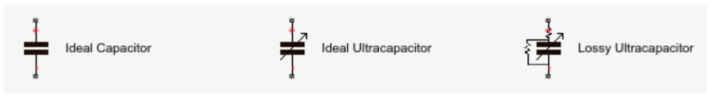
```
v == vc + i*R;
```

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

In the Simscape file, the initial condition (initial voltage in this example) is applied to variable `vc` with `priority = priority.high`, because this is a differential variable. 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 4-42.

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 4-55, 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

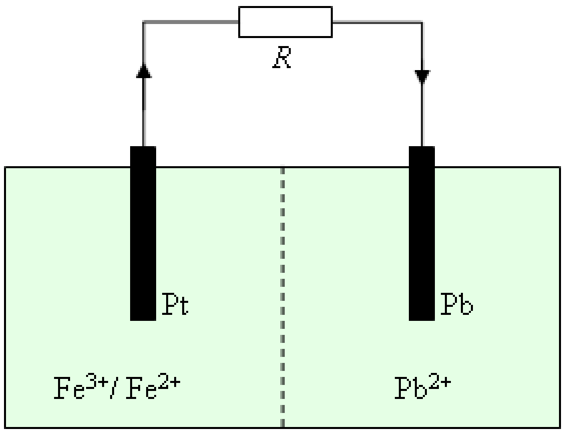
```
MyElectroChem_lib
```

For more information on the library build process, see “Building Custom Block Libraries” on page 4-25.

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 on page 4-69].



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 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`.

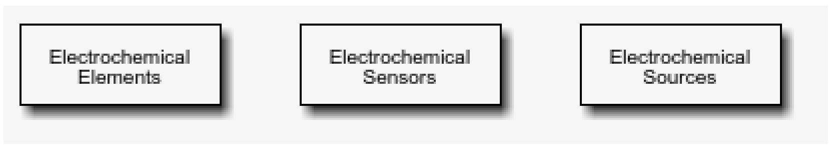
```
domain ElectroChem
% Electrochemical Domain
% Define through and across variables for the electrochemical domain

% Copyright 2008-2014 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 2008-2016 The MathWorks, Inc.

    nodes
        A = ElectroChem.ElectroChem; % A:top
    end

    connections
        connect(A, *);
    end

end
```

The component has one electrochemical port, named A, located at the top of the block icon.

The component uses a connection to an implicit reference node:

```
connect(A, *);
```

For more information on component connections and the implicit reference node syntax, see “Connections to Implicit Reference Node” on page 2-66.

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 2008-2013 The MathWorks, Inc.

    nodes
        A = ElectroChem.ElectroChem; % A:top
        B = ElectroChem.ElectroChem; % B:bottom
    end

    parameters
        mu0 = {0, 'J/mol'}; % Chemical potential
    end

    variables(Access=private)
        ndot = { 0, 'mol/s' }; % Molar flow rate
    end

    branches
        ndot: A.ndot -> B.ndot; % Through variable ndot from node A to node B
    end

    equations
```

```
        let
            mu = A.mu - B.mu; % Across variable from A to B
        in
            mu == mu0;
        end
    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 2008-2013 The MathWorks, Inc.

    nodes
        A = ElectroChem.ElectroChem; % A:top
        B = ElectroChem.ElectroChem; % B:bottom
    end

    outputs
        out  = { 0, 'mol/s' }; % PS:top
    end

    variables(Access=private)
        ndot = { 0, 'mol/s' }; % Molar flow rate
    end

    branches
        ndot: A.ndot -> B.ndot; % Through variable ndot from node A to node B
    end

    equations
        let
            mu = A.mu - B.mu; % Across variable from A to B
        in
            mu == 0;     % No potential drop
            out == ndot; % Equate value of molar flow to PS output
        end
    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 `branches` section and the `let` statement in the equation section 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 second equation assigns the value of the Through variable to the Physical Signal output.

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 2008-2013 The MathWorks, Inc.

    nodes
        A = ElectroChem.ElectroChem; % A:top
        B = ElectroChem.ElectroChem; % B:bottom
    end

    outputs
        out  = { 0, 'J/mol' }; % PS:top
    end

    variables(Access=private)
        ndot = { 0, 'mol/s' }; % Molar flow rate
    end

    branches
        ndot: A.ndot -> B.ndot; % Through variable ndot from node A to node B
    end

    equations
        let
            mu = A.mu - B.mu; % Across variable from A to B
        in
            ndot == 0; % Draws no molar flow
            out == mu; % Equate value of chemical potential difference to PS output
        end
    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 `branches` section and the `let` statement in the equation section 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. The second equation assigns the value of the Across variable to the Physical Signal output.

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 ν 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 :1 :fixed
% Represents a solution of dissolved ions. The port A presents the
% chemical potential defined by mu0 + log(n/(C0*M))*R*T where mu0 is the
% standard state oxidizing potential, n is the number of moles of the ion,
% C0 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 2008-2015 The MathWorks, Inc.

    nodes
        A = ElectroChem.ElectroChem; % A:top
    end

    parameters
        mu0 = {-7.42e+04, 'J/mol'}; % Standard state oxidizing potential
        m_solvent = {1, 'kg'};      % Mass of solvent
        T = {300, 'K'};             % Temperature
    end

    parameters (Access=private)
        R = {8.314472, '(J/K)/mol'}; % Universal gas constant
        C0 = {1, 'mol/kg'};          % Standard concentration
        n1 = {1e-10, 'mol'};         % Minimum number of moles
    end

    variables
        ndot = { 0, 'mol/s' }; % Molar flow rate
        n  = {value = { 0.01, 'mol' }, priority = priority.high}; % Quantity of ions
    end

    branches
        ndot : A.ndot -> *; % Through variable ndot
    end

    equations
        n.der == ndot;
        if n > n1
            A.mu == mu0 + log(n/(C0*m_solvent))*R*T;
        else
            A.mu == mu0 + (log(n1/(C0*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 equation section. Here the Universal Gas constant (`R`) and the Standard Concentration (`C0`) 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, (μ_2 – μ_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 2008-2017 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'   };      % Current
        ndot = { 0, 'mol/s' }; % Molar flow rate
    end

    branches
        i   : p.i    -> n.i;    % Through variable i from node p to node n
        ndot: A.ndot -> B.ndot; % Through variable ndot from node A to node B
    end

    equations
        let
            k = 1/(z*F);
            v = p.v - n.v;    % Across variable v from p to n
            mu = A.mu - B.mu; % Across variable mu from A to B
        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 “Library Configuration Files” on page 4-29.

Using the Custom Components to Build a Model

The “Battery Cell with Custom Electrochemical Domain” 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.

Language Reference

across	Establish relationship between component variables and nodes
annotations	Control appearance of Simscape block based on the component
assert	Program customized run-time errors and warnings
branches	Establish relationship between component Through variables and nodes
component	Component model keywords
components	Declare member components included in composite component
connect	Connect two or more component ports of the same type
connections	Define connections for member component ports in composite component
delay	Return past value of operand
der	Return time derivative of operand
domain	Domain model keywords
edge	Trigger event
entry	Specify actions to be performed upon entering a mode
equations	Define component equations
events	Model discrete events
import	Import model classes
initial	Specify initial mode in mode chart
initialevent	Initialize event variables
inputs	Define component inputs, that is, Physical Signal input ports of block
integ	Perform time integration of expression
intermediates	Define intermediate terms for use in equations
modecharts	Declare mode charts that include operating modes and transitions
modes	Declare operating modes in mode chart
nodes	Define component nodes, that is, conserving ports of block
outputs	Define component outputs, that is, Physical Signal output ports of block
parameters	Specify component parameters
setup	(Not recommended) Prepare component for simulation
tablelookup	Return value based on interpolating set of data points
through	Establish relationship between component variables and nodes
time	Access global simulation time
transitions	Define transitions between modes in mode chart
value	Convert variable or parameter to unitless value with specified unit conversion
variables	Define domain or component variables

across

Establish relationship between component variables and nodes

Syntax

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

Description

Note `across` will be removed in a future release. Use equations instead. For more information, see “Define Relationship Between Component Variables and Nodes” on page 2-21.

`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`

Introduced in R2008b

annotations

Control appearance of Simscape block based on the component

Syntax

```
annotations
   [Id1, Id2] : ExternalAccess=value;
   UILayout = [UIGroup("Title 1",p1,p2) UIGroup("Title 2",p3)]
   Icon = 'filename';
   [port1, port2] : Side=value;
   [param1, param2] : UnitDropdown = common
end
```

Description

`annotations` begins the annotations section, which is terminated by an `end` keyword. The `annotations` section in a component file lets you provide annotations that control various cosmetic aspects of a Simscape block generated from this component.

Use the `annotations` section to:

- Define conditional visibility of component members, such as parameters, variables, nodes, inputs, and outputs, in block icons and dialog boxes. Possible values are: `modify`, `observe`, and `none`.
- Specify block interface layout by defining titled groups of component parameters, the order of these groups, and the order of parameters in each group. When you deploy the component as a custom Simscape block, these groups translate into dialog box tabs (and into Property Inspector tree nodes). `UILayout` is a class-level annotation, meaning that it can appear only once per component file. For more information, see “Group and Reorder Block Parameters Using Annotation” on page 4-37.
- Specify a custom block icon and change it based on the block variant.
- Control port location by placing it on a specific side of the block icon. Ports on a block icon correspond to nodes, inputs, and outputs declared in the underlying component file. Possible values are: `left`, `right`, `top`, and `bottom`.
- Prepopulate a unit drop-down list for a parameter in the block dialog box with commonly used units.

Examples

The following example hides inapplicable parameters from the block dialog box based on the control parameter value.

```
component MyPipe
  parameters
    circular  = true;             % Circular pipe?
    d_in      = { 0.01, 'm' };    % Pipe internal diameter
    area      = { 1e-4, 'm^2' };  % Noncircular pipe cross-sectional area
    D_h       = { 1.12e-2, 'm' }; % Noncircular pipe hydraulic diameter
  end
  if circular
  % Hide inapplicable parameters
    annotations
```

```
        [area, D_h] : ExternalAccess=none;
     end
     equations
        % First set of equations, for circular pipe
     end
   else
   % Hide inapplicable parameter
     annotations
        d_in : ExternalAccess=none;
     end
     equations
        % Second set of equations, for noncircular pipe
     end
   end
   [...] % Other parameters, variables, branches, equations
end
```

The next example exposes a thermal port **H** and changes the customized block icon based on the control parameter value.

```
parameters
    thermal_effects = false; % Model thermal effects?
end
nodes (ExternalAccess=none)
   H = foundation.thermal.thermal;
end
if thermal_effects
  % Use icon with additional thermal port
    annotations
       H : ExternalAccess=modify;
       Icon = 'pipe_thermal.jpg';
    end
end
```

The following example customizes the names and locations of block ports. The block contains two electrical ports, labeled **+** and **-**, located on the left and right sides of the block icon, respectively, and a thermal port **H**, located on the top side.

```
nodes
    H = foundation.thermal.thermal;
    p = foundation.electrical.electrical; % +
    n = foundation.electrical.electrical; % -
end
annotations
    H : Side = top;
    p : Side = left;
    n : Side = right;
end
```

Note You cannot conditionally switch port sides, that is, include `Side` annotations in branches of a conditional statement. For more information, see “Control Port Locations Using Annotations” on page 4-39.

The next example specifies that the drop-down list for the **Gain** parameter includes a list of common units, such as those available in the Simulink-PS Converter and the PS-Simulink Converter block dialog boxes.

```
annotations
    Gain : UnitDropdown = common
```

```
end
```

See Also

`inputs` | `nodes` | `outputs` | `parameters` | `variables`

Topics

“Defining Conditional Visibility of Component Members” on page 2-81
“Attribute Lists” on page 2-103
“Use Advanced Techniques to Customize Block Display” on page 4-47
“Customize the Block Icon” on page 4-42
“Customize the Names and Locations of the Block Ports” on page 4-38
“Physical Signal Unit Propagation”
“How to Specify Units in Block Dialogs”

Introduced in R2019a

assert

Program customized run-time errors and warnings

Syntax

```
assert (predicate_condition, message, Action);
```

Description

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

```
assert (predicate_condition, message, Action);
```

`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.
`Action`	Optional attribute that specifies whether triggering the assert results in a warning or an error during simulation. The default action is error.

The `Action` attribute lets you specify the assert action based on an enumerated parameter value. A built-in enumeration `simscape.enum.assert.action` allows three possible actions when the assertion is triggered: `error`, `warn`, and `none`. You can provide an enumerated value directly to the `Action` attribute:

```
assert(u > 0, Action = simscape.enum.assert.action.warn)
```

or create an enumerated parameter and let the block user control the assert action:

```
parameters
  assert_action = simscape.enum.assert.action.warn
end

equations
  assert(u > 0, Action = assert_action)
end
```

You can use the `assert` construct in:

- The top-level equations, including initial 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.

When you include `assert` constructs in initial equations, their predicate conditions are checked only once, after solving for initial conditions (before the start of simulation, see “Initial Conditions Computation”). Use these assertions to safeguard against the model initializing with nonphysical values.

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 `Action` attribute to `simscape.enum.assert.action.warn`. 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',
                          Action = simscape.enum.assert.action.warn);
```

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

The warning message appears once, at the first time step when the predicate condition is violated. In this example, the warning message appears at the first time step when the pressure drops below vapor level. As long as the pressure stays below that level, the message is not repeated at subsequent time steps. However, as the simulation continues, if the pressure raises above the vapor saturation level and then again drops below that level, the assertion gets reactivated and the warning message appears once again.

User-Controllable Action

If you want to let the block user control the action upon triggering the assert, create an enumerated parameter and set the `Action` attribute to be based on the value of this parameter.

For example, in a Stepper Motor block, you can let the block user decide upon the desired action when the motor slips. Declare a control parameter, based on the built-in assert action enumeration, and add the following `assert` construct to the component equations:

```
parameters
  assert_action = simscape.enum.assert.action.warn % Action on slipping
end

equations
  assert(slipping<1,'Stepper motor slip',Action = assert_action)
end
```

In this case, the default action is also a run-time warning, like in the previous example. However, the block dialog contains an enumerated parameter, **Action on slipping**, with three possible values: `error`, `warn`, `none`. This parameter lets the block user decide whether the simulation should stop with an error, continue with a warning, or ignore the motor slips completely.

Compatibility Considerations

Assert Action Attribute
Behavior changed in R2019a

Prior to R2019a, you specified the assert action by using the `Warn = true|false` attribute. This attribute still works. Currently, there are no plans to remove it.

Internally, the old attribute values are automatically mapped to the appropriate values of the new `Action` attribute:

Old Syntax	New Syntax
`Warn = false`	`Action = simscape.enum.assert.action.error`
`Warn = true`	`Action = simscape.enum.assert.action.warn`

You cannot use the `Warn` and `Action` attributes together in a single `assert` construct. When authoring new components, use the `Action` attribute because it provides more flexibility.

See Also
`equations`

Topics
"Programming Run-Time Errors and Warnings" on page 2-48

Introduced in R2011b

branches

Establish relationship between component Through variables and nodes

Syntax

```
branches a : node1.a -> node2.a; end
```

Description

`branches` begins the branches section, which is terminated by an `end` keyword. This section contains one or more branch statements, which establish the relationship between the Through variables of the component and the domain.

For example, a domain declaration contains a Through variable `a`:

```
variables(Balancing=true)
    a = { 0, 'N' }
end
```

and a component declares two nodes, `node1` and `node2`, associated with this domain, and a variable `a`:

```
variables
    a = { 0, 'N' };
end
```

The name of the component variable does not have to match that of the domain variable, but the units must be commensurate (in this example, `'N'`, `'kg*m/s^2'`, `'lbf'`, and so on).

To establish a connection between the component variable `a` and the domain Through (balancing) variable `a`, write a branch statement, such as:

```
branches
    a : node1.a -> node2.a;
end
```

`node1.a` and `node2.a` identify the conserving equations on `node1` and `node2`, and the component variable `a` is a term participating in those conserving equations. The branch statement declares that `a` flows from `node1` to `node2`. Therefore, `a` is subtracted from the conserving equation identified by `node1.a`, and `a` is added to the conserving equation identified by `node2.a`.

A component can use each conserving equation identifier multiple times. For example, the component declares the following variables and branches:

```
variables
  a1 = { 0, 'N' }
  a2 = { 0, 'N' }
  a3 = { 0, 'N' }
end

branches
  a1 : node1.a -> node2.a;
  a2 : node1.a -> node2.a;
  a3 : node2.a -> node1.a;
end
```

Then, assuming that `node1` and `node2` are not referenced by any other `branch` or `connect` statements, the conserving equations at these nodes are:

- For `node1`

  ```
  - a1 - a2 + a3 == 0
  ```
- For `node2`

  ```
  a1 + a2 - a3 == 0
  ```

The following rules apply:

- Each conserving equation belongs to a node associated with a domain. All variables participating in that conserving equation must have commensurate units.
- A node creates one conserving equation for each of the Through (balancing) variables in the associated domain. Branch statements do not create new equations. They add and subtract terms in the existing conserving equations at the nodes.
- The second and third arguments do not need to be associated with the same domain. For example, one can be associated with a gas domain, and the other with a thermal domain, with the heat flow exchange defined by the branch statement.
- You can replace either the second or the third argument with `*` to indicate the reference node. When you use `*`, the variable indicated by the first argument is still added to or subtracted from the equation indicated by the other identifier, but no equation is affected by the `*`.

Examples

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 `branches` section:

```
branches
   i : p.i -> n.i;
end
```

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

For a grounding component, which has one electrical node `V`, define current `i` as a Through variable flowing from node `V` to the reference node:

```
branches
   i : V.i -> *;
end
```

For a mutual inductor or transformer, with primary and secondary windings, the `branches` section must contain two statements, one for each winding:

```
branches
    i1 : p1.i -> n1.i;
    i2 : p2.i -> n2.i;
end
```

For a component such as a constant volume pneumatic chamber, where you need to establish the heat flow exchange between the pneumatic and the thermal domains, the declaration section contains the two nodes and the heat flow variable:

```
nodes
   A = foundation.pneumatic.pneumatic;
   H = foundation.thermal.thermal;
end
variables
   h = { 0 , 'J/s' };
end
```

and the `branches` section establishes the heat flow exchange between the two domains:

```
branches
   h : A.Q -> H.Q;
end
```

This statement defines the heat flow `h` as a Through variable flowing from the pneumatic node `A`, associated with the chamber inlet, to the thermal node `H`, associated with the thermal mass of gas in the chamber.

See Also

`nodes` | `variables`

Topics

"Define Relationship Between Component Variables and Nodes" on page 2-21

Introduced in R2013b

component

Component model keywords

Syntax

```
component
nodes
inputs
outputs
parameters
variables
components
intermediates
branches
connections
equations
events
annotations
```

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 implementation sections, such as branches, equations, events, and so on. The order of these sections does not matter.

Note The file can contain multiple instances of declaration blocks or implementation sections of the same type, with the exception of the `setup` section. There may be no more than one `setup` section per component. However, starting in R2019a, using `setup` is not recommended. For better alternatives, see “setup is not recommended” on page 5-69.

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 “Declare Component Nodes” on page 2-14 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 on page 2-5. See “Declare Component Inputs and Outputs” on page 2-16 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 on page 2-5. See “Declare Component Inputs and Outputs” on page 2-16 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 on page 2-5. See “Declare Component Parameters” on page 2-11 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 will appear on the **Variables** tab of 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 on page 2-5. See “Declare Component Variables” on page 2-7 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 2-59 for more information.
- `intermediates` begins a declaration block of named intermediate terms, which is terminated by an `end` keyword. This block contains declarations of intermediate terms that can be reused in any `equations` section of the same component or of an enclosing composite component. See “Using Intermediate Terms in Equations” on page 2-35 for more information.

`branches` begins the branches section, which is terminated by an `end` keyword. This section establishes relationship between the Through variables of the component and the domain. Relationship between the Across variables is established in the equation section. See “Define Relationship Between Component Variables and Nodes” on page 2-21 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 2-64 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 2-24 for more information.

`events` begins the events section, which is terminated by an `end` keyword. This section manages the event updates. See “Discrete Event Modeling” on page 2-52 for more information.

`annotations` begins the annotations section, which is terminated by an `end` keyword. This section lets you provide annotations in a component file that control various cosmetic aspects of a Simscape block generated from this component. See `annotations` for more information.

Table of Attributes

For component model attributes, as well as declaration member attributes, see “Attribute Lists” on page 2-103.

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
  branches
    t : r.t -> c.t;
  end
  equations
    assert(k>0)
    w == r.w - c.w;
    t == k * theta;
    w == theta.der;
  end
end
```

See Also
`domain`

Topics
"Creating Custom Components" on page 1-13

Introduced in R2008b

components

Declare member components included in composite component

Syntax

```
components(ExternalAccess=observe)
    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 `ExternalAccess` attribute value set to `observe` (for more information on member attributes, see “Attribute Lists” on page 2-103).

Each member component is defined by assignment to an existing component file. See “Declaring Member Components” on page 2-59 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 “Building Custom Block Libraries” on page 4-25.

```
components(ExternalAccess=observe)
    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 2-64 for more information.

Once you declare a member component, you can use its parameters and variables in the equation 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 2-60 for more information.

You can also use `for` loops to declare an array of member components and specify the component connections. For more information, see “Component Arrays” on page 3-29.

Examples

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

```
components(ExternalAccess=observe)
    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`

Topics

"Declaring Member Components" on page 2-59

Introduced in R2012b

connect

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.

The * symbol indicates a connection to an implicit reference node:

```
connect(n1, *);
```

For more information, see “Connections to Implicit Reference Node” on page 2-66.

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,

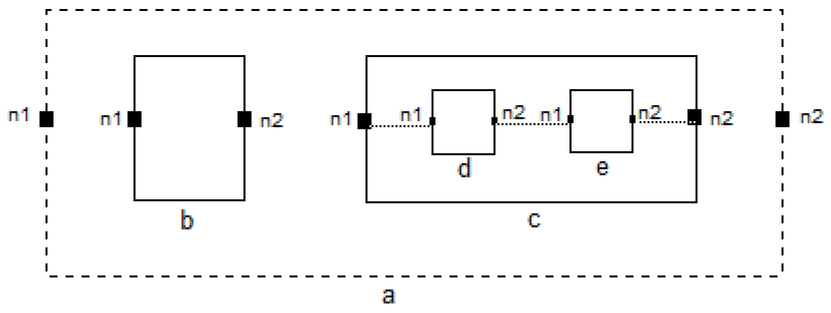
```
connect(s, d1, d2);
```

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

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

You can also use `for` loops to declare an array of member components and specify the component connections. For more information, see “Component Arrays” on page 3-29.

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(ExternalAccess=observe)
    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`

Topics

"Specifying Component Connections" on page 2-64

Introduced in R2012b

connections

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 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.

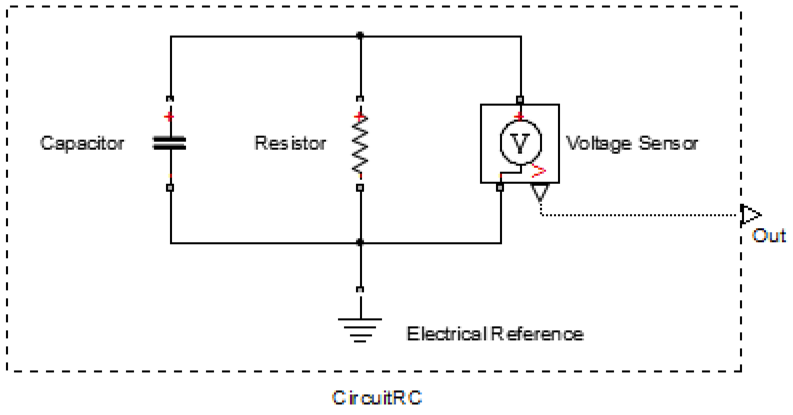
```
connections
   connect(a, c1.a);
   connect(c1.b, c2.a);
   connect(c2.b, b);
end
```

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

You can also use `for` loops to declare an array of member components and specify the component connections. For more information, see "Component Arrays" on page 3-29.

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.



```
component CircuitRC
   outputs
     Out = { 0.0, 'V' }; % I:right
   end
   parameters
      p1 = {1e-6, 'F'};  % Capacitance
      p2 = {10, 'Ohm'};  % Resistance
   end
   components(ExternalAccess=observe)
      c1 = foundation.electrical.elements.capacitor(c=p1);
      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`

Topics

"Declaring Member Components" on page 2-59

“Specifying Component Connections” on page 2-64

Introduced in R2012b

delay

Return past value of operand

Syntax

```
delay(u,tau)
delay(u,tau, History = u0, 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 = u0, 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 `u0` must be commensurate. The default `u0` 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 **Source code** 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 **Simscape > View source code**.

See Also

`equations`

Introduced in R2012a

der

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:

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

`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.

You can nest `der` operators to specify higher order derivatives. For example, `der(der(x))`is the second order time derivative of x.

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.
- 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`

Introduced in R2008b

domain

Domain model keywords

Syntax

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

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 on page 2-5. See “Declare Through and Across Variables for a Domain” on page 2-6 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 2-98 for more information.

`intermediates` begins a declaration block of named intermediate terms, which is terminated by an `end` keyword. This block contains declarations of intermediate terms that can be reused in equations of components that have nodes of this domain type. This block is optional.

See “Using Intermediate Terms in Equations” on page 2-35 for more information.

Table of Attributes

For declaration member attributes, see “Attribute Lists” on page 2-103.

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`

Topics

“When to Define a New Physical Domain” on page 1-11
“Foundation Domain Types and Directory Structure” on page 6-2

Introduced in R2008b

edge

Trigger event

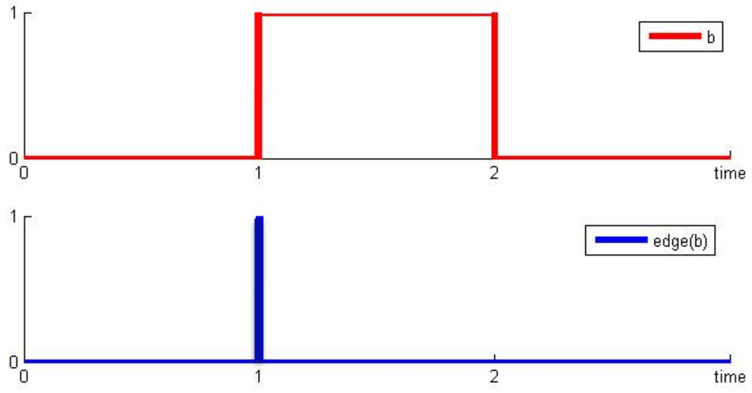
Syntax

`edge(b)`

Description

`edge(b)` takes a scalar boolean expression `b` as input. It returns true, and triggers an event, when and only when the input argument changes value from false to true. The return data type of `edge` is `event`. Event data type is a special category of boolean type, which returns true only instantaneously, and returns false otherwise.

The following graphic illustrates the difference between boolean and event data types.



`edge(b)` returns true only when `b` changes from false to true.

You use the `edge` operator to define event predicates in `when` clauses. For more information, see `events`.

Examples

`edge(b)` returns true when `b` changes from false to true, that is, triggers an event on the rising edge of condition `b`.

To trigger an event on the falling edge of condition `b`, use `edge(~b)`.

To trigger an event both on the rising edge and on the falling edge of condition `b`, use `edge(b)||edge(~b)` as the event predicate in the `when` clause. For more information on data derivation rules between boolean and event data types, see “Event Data Type and edge Operator” on page 2-52.

To trigger an event at a specific time, for example, 2 seconds after the start of simulation, use `edge(time>{2.0,'s'})`.

See Also

`events` | `initialevent`

Topics

"Discrete Event Modeling" on page 2-52
"Triggered Delay Component" on page 2-55
"Enabled Component" on page 2-56

Introduced in R2016a

entry

Specify actions to be performed upon entering a mode

Syntax

```
entry v_old = v; end
```

Description

The `entry` block, terminated by an `end` keyword, is an optional section inside a mode declaration that lets you specify the actions to be performed upon entering the mode. These actions are event variable updates based on the value of a continuous expression immediately before entering the mode.

```
modes
    mode m
       entry
          v_old = v;
       end
       equations
       ...
       end
    end
end
```

The `entry` section is especially useful for modeling state reset because, in the majority of state reset use cases, the reset value is a function of the previous value of the variable. For example, when modeling a slider moving between two hard stops, the new velocity depends on the velocity before impact. For more information and an example, see “State Reset Example” on page 3-11.

In each entry action, the left-hand side must be an event variable. The right-hand side is a continuous expression, evaluated immediately before entering the mode. This expression can include any combination of continuous variables, event variables, and intermediates.

You can use entry actions both in instantaneous and regular modes.

See Also

`initial` | `modecharts` | `modes` | `transitions`

Topics

“State Reset Modeling” on page 3-11
“Mode Chart Modeling” on page 3-2

Introduced in R2020b

equations

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. 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 equation section of a Simscape file is executed throughout the simulation. You can also specify equations that are executed during model initialization only, by using the `(Initial=true)` attribute. For more information, see “Initial Equations” on page 5-37.

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 a valid MATLAB expression. `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
       EquationList
   { elseif Expression
       EquationList }
   else
       EquationList
   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 declare intermediate terms in the `intermediates` section of a component or domain file and then use these terms in any equations section in the same component file, in an enclosing composite component, or in a component that has nodes of that domain type.

You can also define intermediate terms directly 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.

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 2-25)
  - 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
- 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.

All arguments that specify size or dimension must be unitless constants or unitless compile-time parameters.

Supported Functions

Name	Restrictions	Discontinuous
`ones`		
`zeros`		
`cat`		
`horzcat`		
`vertcat`		
`length`		
`ndims`		
`numel`		
`size`		
`isempty`		
`isequal`		Possibly, if arguments are real and have the same size and commensurate units
`isinf`		Yes
`isfinite`		Yes
`isnan`		Yes
`plus`		
`uplus`		
`minus`		
`uminus`		
`mtimes`		
`times`		
`mpower`		
`power`		
`mldivide`	First argument must be a scalar	
`mrdivide`	Second argument must be a scalar	
`ldivide`		
`rdivide`		
`mod`		Yes
`sum`		
`prod`		
`floor`		Yes
`ceil`		Yes
`fix`		Yes
`round`		Yes

Name	Restrictions	Discontinuous
`eq`	Do not use with continuous variables	
`ne`	Do not use with continuous variables	
`lt`		
`gt`		
`le`		
`ge`		
`and`		Yes
`or`		Yes
`logical`		Yes
`sin`		
`cos`		
`tan`		
`asin`		
`acos`		
`atan`		
`atan2`		Yes
`log`		
`log10`		
`sinh`		
`cosh`		
`tanh`		
`exp`		
`sqrt`		
`abs`		Yes
`sign`		Yes
`any`		Yes
`all`		Yes
`min`		Yes
`max`		Yes
`double`		
`int32`		Yes
`uint32`		Yes
`repmat`		
`reshape`	Expanded empty dimension is not supported	

Name	Restrictions	Discontinuous
`dot`		
`cross`		
`diff`	In the two argument overload, the upper bound on the second argument is 4, due to a Simscape limitation	

Initial Equations

The `(Initial=true)` attribute lets you specify equations that are executed during model initialization only:

```
equations (Initial=true)
   Expression1 == Expression2;
end
```

The default value of the `Initial` attribute for equations is `false`, therefore you can omit this attribute when declaring regular equations.

For more information on when and how to specify initial equations, see “Initial Equations” on page 2-31.

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
```

If a function has multiple return values, use it in a `let` statement to access its values. For example:

```
equations
  let
    [m, i] = min(a);
  in
    x == m;
    y == i;
    end
end
```

See Also

`assert` | `delay` | `der` | `function` | `integ` | `intermediates` | `tablelookup` | `time`

Topics

"Defining Component Equations" on page 2-24
"Using Conditional Expressions in Equations" on page 2-33
"Using Intermediate Terms in Equations" on page 2-35

Introduced in R2009a

events

Model discrete events

Syntax

```
events
  when EventPredicate
    AssignmentList
  end
end
```

Description

`events` begins the events section, which is terminated by an `end` keyword. The `events` section in a component file manages event updates. It is executed throughout the simulation.

The `events` section can contain only `when` clauses.

The `when` clause serves to update the values of the event variables. The syntax is

```
when EventPredicate
  var1 = expr1;
  var2 = expr2;
  ...
end
```

`EventPredicate` is an expression that defines when an event occurs. It must be an expression of event data type, as described in “Event Data Type and edge Operator” on page 2-52.

The variables in the body of the `when` clause must be declared as event variables on page 2-52. When the event predicate returns true, all the variables in the body of the `when` clause simultaneously get updated to the new values.

A `when` clause can optionally have one or more `elsewhen` branches:

```
when EventPredicate
  var1 = expr1;
  var2 = expr2;
  ...
elsewhen EventPredicate
  var1 = expr3;
  ...
end
```

Note The default `else` branch in a `when` clause is illegal.

The following rules apply:

- The order of `when` clauses does not matter.
- The order of the variable assignments in the body of a `when` clause does not matter because all updates happen simultaneously.

- A `when` clause cannot update an event variable more than once within the same assignments list.
- Two independent `when` clauses may not update the same event variable. You must use an `elsewhen` branch to do this.
- The order of `elsewhen` branches matters. If multiple predicates become true at the same point in time, only the branch with the highest precedence is activated. The precedence of the branches in a `when` clause is determined by their declaration order. That is, the `when` branch has the highest priority, while the last `elsewhen` branch has the lowest priority.

See Also
`edge` | `initialevent`

Topics
"Discrete Event Modeling" on page 2-52
"Triggered Delay Component" on page 2-55
"Enabled Component" on page 2-56

Introduced in R2016a

function

Reuse expressions in component equations and in member declarations of domains and components

Syntax

```
function out = FunctionName(in1,in2)
   definitions
      out = Expression1(in1,in2);
   end
end
```

Description

The `function` keyword begins the Simscape function declaration, which is terminated by an `end` keyword.

The keyword `function` must be followed by the function header, which includes the function name, inputs, and outputs.

The body of the function must be enclosed inside the `definitions` block, which is terminated by an `end` keyword. The `definitions` block contains equations that express the output arguments of the function in terms of its input arguments. This block is required.

The following syntax declares a simple function.

```
function out = FunctionName(in1,in2)
   definitions
      out = Expression1(in1,in2);
   end
end
```

If the function has multiple return values, the syntax is:

```
function [out1,out2] = FunctionName(in1,in2)
```

Depending on whether the Simscape function is a main or local function:

- Main function — You must place the function declaration in a file of the same name with a file name extension of `.ssc`. The file name must match the function name. For example, function `foo` must be in a file called `foo.ssc`. The file must begin with the `function` keyword. Only blank lines and comments can precede `function`.
- Local function — Include the function declaration in a component, domain, or function file, after the final `end` keyword that concludes the description of the component, domain, or main function. The local function is accessible only by that component, domain, or main function.

Syntax Rules

- One or more output parameters are allowed.
- If an output parameter is not used on the left-hand side of the `definitions` section, you get an error.

- Zero or more input parameters are allowed.
- When the function is called, the number of input arguments must match the number of input parameters.
- Input parameters are positional. This means that the first input argument during the function call is passed to the first input parameter, and so on. For example, if you write an equation:

  ```
  o == FunctionName(5,2);
  ```

  then `in1` is 5 and `in2` is 2.
- If the function has multiple return values, they are also positional. That is, the first output parameter gets assigned to the first return value, and so on.
- If the function has multiple return values, the rules and restrictions are the same as for declaration functions. For more information, see “Multiple Return Values” on page 3-22.
- The `definitions` section can contain intermediate terms and `if-elseif-else` statements. The same syntax rules as in the declaration section of a `let` statement apply. For more information, see “Using Intermediate Terms in Equations” on page 2-35.
- The `definitions` section cannot contain expressions with dynamic semantics, such as `integ`, `time`, `der`, `edge`, `initialevent`, or `delay`.

Packaging Rules for Function Files

- Simscape function files can reside directly on MATLAB path or in package directories. For more information, see “Organizing Your Simscape Files” on page 4-25.
- You can use source protection, as described in “Using Source Protection for Simscape Files” on page 4-26.
- Importing a package imports all the Simscape functions in this package. For more information, see “Importing Domain and Component Classes” on page 2-108.
- If a MATLAB function and a Simscape function have the same name, the MATLAB function has higher precedence.

Examples

Declare a function that computes the square of a sum of two numbers:

```
function out = SumSquared(in1,in2)
   definitions
      out = in1^2 + 2*in1*in2 + in2^2;
   end
end
```

Save the function in a file named `SumSquared.ssc`, on the MATLAB path.

This component calls the `SumSquared` function to compute the square of a sum of its parameters `p1` and `p2`.

```
component MyComp
   outputs
      o = 0;
   end
   parameters
      p1 = 5;
      p2 = 2;
```

```
   end
   equations
      o == SumSquared(p1,p2);
   end
end
```

For a more detailed example of declaring and using a main Simscape function, see the “Simscape Functions” example.

For a detailed example of declaring and using a local Simscape function, see “Local Simscape Functions” on page 3-27.

See Also

`equations`

Topics

“Simscape Functions” on page 3-25

Introduced in R2017b

import

Import model classes

Syntax

```
import package_or_class;
```

```
import package.*;
```

Description

The `import` statements allow access to model class or function names defined in other scopes (packages) without a fully qualified reference. You must place `import` statements at the beginning of a Simscape file.

There are two types of `import` statement syntaxes. One is a qualified import, which imports a specific package, class, or function:

```
import package_or_class;
```

The other one is an unqualified import, which imports all subpackages, classes, and functions under the specified package:

```
import package.*;
```

The package or class name must be a full path name, starting from the library root (the top-level package directory name) and containing subpackage names as necessary.

Import statements are subject to the following constraints:

- The imported name must be a full path name, starting from the library root, even if the `import` statement is used in a component class defined under the same package as the domain or component class that is being imported.
- You must place `import` statements at the beginning of a Simscape file. The scope of imported names is the entire Simscape file, except the `setup` section.
- In qualified import statements, the imported name can refer to a subpackage, a model class (domain class or component class), or a function. For example, in the `import A.B.C;` statement, `C` can be either a subpackage name, a class name, or a function name. In unqualified import statements, the imported name must refer to a package or subpackage. For example, in the `import A.B.*;` statement, `B` must be a subpackage name (of package `A`).
- It causes a compilation error if an unqualified imported name is identical to other names within the same scope, provided the duplicate name is in use. For example, assume that subpackages `A.B` and `A.B1` both contain a component class `C`. The following code:

  ```
  import A.B.C;
  import A.B1.*;
  component M
     components (ExternalAccess=observe)
       c = C;
     end
  end
  ```

causes a compile-time error. However, the following code is legal (provided that D is defined only in A.B) because C is not used:

```
import A.B.C;
import A.B1.*;
component M
   components (ExternalAccess=observe)
     d = D;
   end
end
```

This code is also legal:

```
import A.B;
import A.B1;
component M
   components
     c1 = B.C;
     c2 = B1.C;
   end
 end
```

because you import two different names into this scope (`B` and `B1`), and access the two different component classes `C` through their parent classes `B` and `B1`.

Examples

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

```
import foundation.electrical.electrical;  % electrical domain class definition
import foundation.electrical.elements.*;  % all electrical elements
component ParElResistors
  nodes
     p = electrical;
     n = electrical;
  end
  parameters
    p1 = {3 , 'Ohm'};
  end
  components(ExternalAccess=observe)
    r1 = resistor(R=p1);
    r2 = resistor(R=p1);
    r3 = 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

Topics
"Importing Domain and Component Classes" on page 2-108

Introduced in R2013b

initial

Specify initial mode in mode chart

Syntax

```
initial init_mode : predicate_condition end
```

Description

`initial` begins the initial mode construct in a mode chart. The construct is terminated by an `end` keyword. It contains one statement with the following syntax:

```
init_mode : predicate_condition
```

where:

- *`init_mode`* is the mode active at the start of simulation if the expression in the *`predicate_condition`* is true.
- If *`predicate_condition`* is false, then the first mode listed in the `modes` section is active at the start of simulation.

The *`predicate_condition`* must be a parametric expression because it is evaluated at compile time. Using a variable in a predicate results in a compile-time error.

The `initial` construct is optional. If a mode chart does not contain an `initial` construct, then the first mode listed in the `modes` section is active at the start of simulation.

For example, a mode chart declares three modes, `m1`, `m2`, and `m3`:

```
modes
    mode m1
    ...
    end
    mode m2
    ...
    end
    mode m3
    ...
    end
end
```

By default, mode `m1` is active at the start of simulation. If you include the following `initial` construct:

```
initial
    m2 : p1
end
```

then, if the `p1` predicate is true, simulation starts in mode `m2`; otherwise, in mode `m1`.

The `initial` construct can have multiple initial mode statements, for example:

```
initial
    m2 : p1
    m3 : p2
end
```

In this case:

- If the `p1` predicate is true, simulation starts in mode `m2`.
- If the `p2` predicate is true, simulation starts in mode `m3`.
- If both predicates are true, simulation starts in mode `m2` (the first one listed in the `initial` section).
- If both predicates are false, simulation starts in mode `m1` (the first one listed in the `modes` section).

At initialization time, the solver sets the initial mode first, and then checks the transitions. If a transition predicate is true at initialization time, the system might start in a different mode than that specified by the `initial` construct. For example, consider a mode chart that declares three modes, `m1`, `m2`, and `m3`, and defines the following transitions and initial modes:

```
transitions
    m3 -> m2 : p1
end
initial
    m2 : p2
    m3 : p3
end
```

If predicates `p1` and `p3` are both true at initialization time, the solver sets `m3` as the initial mode, but the system immediately transitions from mode `m3` to mode `m2`, and simulation starts in mode `m2`.

See Also

`modecharts` | `modes` | `transitions`

Topics
"Mode Chart Modeling" on page 3-2
"Switch with Hysteresis" on page 3-5

Introduced in R2017a

initialevent

Initialize event variables

Syntax

```
events
  when initialevent
    AssignmentList
  end
end
```

Description

`initialevent` lets you specify initial values of event variables at the start of simulation. The return data type of `initialevent` is `event`, as described in “Event Data Type and edge Operator” on page 2-52. It returns true once during simulation, right after initialization of continuous variables is finished.

The `initialevent` keyword is valid only inside a `when` clause predicate.

Examples

The Asynchronous Sample & Hold block in the Simscape Foundation library initializes the event variable `y_held`, which holds the sampled signal, by using a block parameter.

This example implements an asynchronous sample and hold block where the `y_held` event variable is initialized based on the value of the input physical signal `IC` at the start of simulation.

```
component ASHold
% Asynchronous Sample and Hold

inputs
   IC = {0.0, '1'}; % :left
    U = {0.0, '1'}; % :left
    T = {0.0, '1'}; % :left
end;

outputs
    Y = {0.0, '1'}; % :right
end;

variables (Event = true, Access = private)
     y_held = {value = {0.0, '1'}, priority = priority.high};
end

equations
    Y == y_held;
end

events
    when initialevent
       y_held = IC;
    elsewhen edge(T > 0)
       y_held = U;
    end
end

end
```

See Also

`edge` | `events`

Topics

"Discrete Event Modeling" on page 2-52

Introduced in R2017b

inputs

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

Syntax

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

```
inputs in1; 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 can be defined as:

- A value with unit on page 2-5, where `value` can be a scalar, vector, or matrix. For a vector or a matrix, all signals have the same unit.
- An untyped identifier, to facilitate unit propagation.

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

The following syntax declares 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
```

If you declare an input without a value and unit, as an untyped identifier, it propagates the signal type (size and unit) based on the component connections in the model. Use the following syntax to declare a component input, `in1`, as an untyped identifier.

```
inputs
    in1;
end
```

Note During `ssc_build` validation, or when an input is unconnected in a model, untyped inputs receive the type of unitless scalar, that is, `{0, '1'}`. Therefore, a component with an untyped input must support the type of the input being resolved to unitless scalar.

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
```

You can also reference component parameters in input declarations. For example, you can control the signal size by using a block parameter:

```
component MyTransformer
     parameters
         N = 3; % Number of windings
     end
     inputs
         I = {zeros(N, 1), 'A'};
     end
     ....
 end
```

The following example declares an input port `I` as an untyped identifier. The unit and size of the input physical signal at port I are propagated from the connected output port.

```
 inputs
   I;
 end
```

See Also

`nodes` | `outputs`

Topics

"Declare Component Inputs and Outputs" on page 2-16
"Physical Signal Unit Propagation"

Introduced in R2008b

integ

Perform time integration of expression

Syntax

`integ(expr,t_L)`

Description

The `integ` operator lets you perform time integration of an expression in the `equations` section of a Simscape file without declaring and initializing extra variables.

The full syntax is:

`integ(expr,t_L)`

where:

- `expr` is a Simscape language expression.
- `t_L` is the lower integration limit, specified as a delay time relative to the current time. This operand is optional.

The upper integration limit is the current simulation time. If you omit the lower limit, the integration starts from the simulation start time.

`expr` can be of any type. It will automatically be converted to a `double`.

The following restrictions apply:

- `expr` cannot contain `delay` or `der` operators. Any time-dependency in `expr` is attributed to the integration variable.
- `expr` is assumed to have zero history for times prior to start of simulation.
- `t_L` must be a scalar nonnegative constant or parametric expression with the unit of time.

The return unit of `integ` is the unit of its operand multiplied by a unit of time.

Examples

Calculate the total energy through an electrical branch:

```
e == integ(v*i);
```

Calculate a moving average of the input signal:

```
component MovingAvg
  inputs
    u = 0;
  end
  outputs
    avg = 0;
  end
  parameters
```

```
    T = { 1, 's' };   % Time interval
  end
  equations
    avg == integ(u,T)/T;
  end
end
```

The block generated from this component outputs the moving average of the input signal over a time interval specified by the **Time interval** parameter.

See Also

`equations`

Introduced in R2016a

intermediates

Define intermediate terms for use in equations

Syntax

```
intermediates int_term1 = expr1; end
```

Description

`intermediates` begins an intermediates declaration block, which is terminated by an `end` keyword. In a component file, this block contains declarations of named intermediate terms for use in equations. You can reuse these intermediate terms in any equations section of the same component or of an enclosing composite component.

You can also include an `intermediates` section in a domain file and reuse these intermediate terms in any component that has nodes of that domain type.

When an intermediate term is used in an equation, it is ultimately substituted with the expression that it refers to. Think of an intermediate term as of defining an alias for an expression.

```
intermediates
    int_term1 = expr1;
end
```

Declaring intermediate terms helps with code reuse and readability. For example, if you declare hydraulic diameter and critical Reynolds number as intermediate terms in a hydraulic component, you can then use these terms throughout the component equations.

You can also specify a descriptive name for an intermediate term, as a comment, similar to the way you do it for parameters and variables:

```
intermediates
    int_term1 = expr1; % Descriptive name
end
```

Then, if you include the intermediate term in logged simulation data, this descriptive name appears in the Simscape Results Explorer.

Examples

This example declares the intermediate term `D_h` (hydraulic diameter) as a function of the orifice area:

```
intermediates
    D_h  = sqrt( 4.0 * A / pi ); % Hydraulic diameter
end
```

This example declares the same intermediate term `D_h`, but prevents it from appearing in simulation data logs:

```
intermediates(ExternalAccess = none)
    D_h  = sqrt( 4.0 * A / pi );
end
```

See Also

equations

Topics

"Using Intermediate Terms in Equations" on page 2-35

Introduced in R2018b

modecharts

Declare mode charts that include operating modes and transitions

Syntax

```
modecharts mc1 = modechart ... end end
```

Description

`modecharts` begins a mode charts declaration block, which is terminated by an `end` keyword. `modecharts` is a top-level section in a component file. It can contain one or more `modechart` constructs. Each `modechart` construct declares one mode chart. A mode chart declaration must describe a complete set of operating modes and transition rules between these modes.

For example, the following syntax declares two mode charts, `mc1` and `mc2`.

```
modecharts (ExternalAccess = observe)
    mc1 = modechart
    ...
    end
    mc2 = modechart
    ...
    end
end
```

`modechart` is a named construct. It is terminated by an `end` keyword. A `modechart` construct contains a complete textual representation of the mode chart: modes, transitions, and an optional initial mode specification. If you omit the initial mode specification, then the first mode listed in the `modes` section is active at the start of simulation.

```
modecharts (ExternalAccess = observe)
     mc1 = modechart
        modes
           ...
      end
        transitions
           ...
        end
        initial
           ...
        end
     end
end
```

A mode chart is defined within the scope of its parent component. In other words, its equations and predicates reference the component members, such as parameters and variables.

Member Accessibility Attribute Values

A `modecharts` declaration block has the following attributes:

- `Access` — Defines the read and write access.

- `ExternalAccess` — Sets the visibility in the user interface.

A mode chart cannot be modifiable in the user interface. Therefore, the following rules apply:

- A `modecharts` declaration block can have its `ExternalAccess` attribute set to `observe` or `none`, but not to `modify`.
- The default `Access` attribute value is `public`, and the corresponding default value for the `ExternalAccess` attribute is `modify`. Therefore, if you do not set the `Access` attribute for a `modecharts` declaration block, you must explicitly set its `ExternalAccess` attribute to `observe` or `none`. For example:

  ```
  modecharts (ExternalAccess = observe)
     ...
  end
  ```
- If you set the `Access` attribute to `private` or `protected`, then the default value for the `ExternalAccess` attribute is `observe`. Therefore, you do not have to set the `ExternalAccess` attribute value explicitly, for example:

  ```
  modecharts (Access = protected)
     ...
  end
  ```

  Set the `ExternalAccess` attribute to `none` if you do not want the mode chart to be visible anywhere outside the language. For example:

  ```
  modecharts (Access = private,ExternalAccess = none)
     ...
  end
  ```

See Also
`entry` | `initial` | `modes` | `transitions`

Topics
"Mode Chart Modeling" on page 3-2
"Switch with Hysteresis" on page 3-5

Introduced in R2017a

modes

Declare operating modes in mode chart

Syntax

`modes mode m1 ... end mode m2 ... end end`

Description

`modes` begins a modes declaration block in a mode chart. The `modes` block, terminated by an `end` keyword, can contain one or more `mode` constructs. Each `mode` construct declares one mode.

For example, the following syntax declares two modes, `m1` and `m2`.

```
modes
    mode m1
    ...
    end
    mode m2
    ...
    end
end
```

`mode` is a named construct. It is terminated by an `end` keyword. Each mode declaration contains a complete set of equations that describe this operating mode.

```
modes
    mode m1
       equations
       ...
       end
    end
    mode m2
       equations
       ...
       end
    end
end
```

For every mode, the total number of equation expressions, their dimensionality, and their order must be the same. This restriction is the same as for the equations in different branches of the `if-elseif-else` statement.

This restriction does not apply to the `assert` expressions, because they are not included in the expression count.

A mode declaration can contain an `entry` section, which lets you specify the actions to be performed upon entering the mode. These actions are event variable updates based on the value of a continuous variable immediately before entering the mode.

```
modes
    mode m1
       equations
```

```
         ...
         end
      end
      mode m2
         entry
         ...
         end
         equations
         ...
         end
      end
end
```

The `entry` section is separate from the `equations` section, and the event variable updates in the `entry` section are not counted in the number of equation expressions for the mode.

To implement a state reset, mode charts can contain instantaneous modes and compound transitions. An instantaneous mode is a mode that is active only for one event iteration. You declare instantaneous modes the same way as regular modes, using the same syntax. The same mode can be used as an instantaneous mode in one case and a regular mode in another, depending on the transitions declared in a mode chart. To specify that a mode is instantaneous, list it as the middle mode in a compound transition.

See Also

`entry` | `initial` | `modecharts` | `transitions`

Topics
"Mode Chart Modeling" on page 3-2
"Switch with Hysteresis" on page 3-5

Introduced in R2017a

nodes

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 “Declare Component Nodes” on page 2-14 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 “Building Custom Block Libraries” on page 4-25.

```
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`

Topics

"Declare Component Nodes" on page 2-14
"Define Relationship Between Component Variables and Nodes" on page 2-21

Introduced in R2008b

outputs

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

Syntax

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

```
outputs out1; 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 can be defined as:

- A value with unit on page 2-5, where `value` can be a scalar, vector, or matrix. For a vector or a matrix, all signals have the same unit.
- An untyped identifier, to facilitate unit propagation.

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
```

If you declare an output without a value and unit, as an untyped identifier, then the output signal type (size and unit) is based on the input signal type and unit propagation rules. Use the following syntax to declare a component output, `out1`, as an untyped identifier.

```
outputs
    out1;
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
    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
```

You can also reference component parameters in output declarations. For example, you can control the signal size by using a block parameter:

```
component MyComp
     parameters
         N = 3; % Matrix size
     end
     outputs
         v = {zeros(N), 'm/s'};
     end
     ....
end
```

The following example declares an input port `I` and output port `O` as untyped identifiers. In the block diagram, the output port will be located on the right side of the block icon. The block propagates the unit and size of the physical signal from port `I` to port `O`. For more information, see “Physical Signal Unit Propagation”.

```
inputs
  I;
end
outputs
  O; % :right
end
```

See Also

`inputs` | `nodes`

Topics

“Declare Component Inputs and Outputs” on page 2-16

Introduced in R2008b

parameters

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 on page 2-5. 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
    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`

See Also

`value`

Topics

"Declare Component Parameters" on page 2-11

Introduced in R2008b

setup

(Not recommended) Prepare component for simulation

Note `setup` is not recommended. For more information, see “Compatibility Considerations”.

Syntax

```
function setup
[...]
end
function setup %#simple
[...]
end
```

Description

```
function setup
[...]
end
```

The setup section of a Simscape file 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.

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

The following rules apply:

- The `setup` function is executed as regular MATLAB code.
- All parameters and variables 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
```

- You can use variable names only on the left-hand side of the assignments in the `setup` section. Parameter names can be used on either side.
- All parameters and variables that are externally writable are writable within setup.
- In case of conflict, assignments in the `setup` section override those made in the declaration section. To ensure proper block operation, if you assign a value to a member in the `setup` section, declare this member with an attribute that prevents it from appearing in the block dialog box, such as `(ExternalAccess=observe)`. Otherwise, the assignment made in the `setup` section will override the values specified in the dialog box by the block user. See "Attribute Lists" on page 2-103 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".
- Persistent and global variables are not supported. For more information, see "Persistent Variables" and "Global Variables".
- MATLAB system commands using the `!` operator are not supported.
- `try-end` and `try-catch-end` constructs are not supported.
- Nested functions 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`*`'))`.

Simple Setup

In general, you cannot designate a block parameter as run-time if the underlying component uses it in the setup function. However, if the setup is restricted to simple operations like error-checking, you can declare the `setup` function as simple:

```
function setup %#simple
[...]
end
```

In this case, many of the parameters used in the `setup` function can be designated as run-time parameters.

When you declare `setup` function as simple, the following rules apply:

- All expressions used in a simple `setup` function must restrict themselves to those supported elsewhere in Simscape language. For a complete list of supported functions, see `equations`.
- A value, parameter or variable, may be assigned to only once on any given path through the `setup` function.
- All reads from a parameter must appear after it is assigned in a `setup` function.
- All assignments must end in a semicolon.

- All members that are assigned to must be private parameters or variables of the current component. Simple setup cannot assign to members of child components or members of a base class.
- You can declare local MATLAB variables in a simple `setup` function, but these variables cannot be structures.
- Arguments of `error` and `warning` functions must be literal strings.

In general, making a `setup` function simple means that all parameters are run-time capable. The exception are those parameters that drive conditional assignment:

```
if p1 > 0
  p3 = f1(p2);
else
  p3 = f2(p2);
end
```

In this case, `p1` must be compile-time. However, only those parameters that affect conditional assignment are compile-time. Those that affect error conditions are run-time capable.

Examples

The following `setup` function validates parameters using an `if` statement and the `error` function.

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

To avoid using `setup`, rewrite this example as follows:

```
component MyComponent
   parameters
      LowerThreshold = {1, 'm' };
      UpperThreshold = {1, 'm' };
   end
   [...]
   equations
      assert(LowerThreshold<UpperThreshold,'LowerThreshold is greater than UpperThreshold');
      [...]
   end
   [...]
end
```

Compatibility Considerations

setup is not recommended

Not recommended starting in R2019a

Starting in R2019a, run-time capable domain parameters have been implemented. Unlike component parameters, domain parameters propagate to other components connected to the circuit. Therefore, when you set the parameter as `Run-time` in the source component, it is possible that another component connected to the same circuit is using this parameter in the context which prevents it from being run-time configurable. For example, if one of the components connected to the circuit uses a domain parameter in its `setup` function, you get an error when trying to simulate the model.

There are no plans to remove `setup` at this time. However, to avoid errors with run-time domain parameters, it is recommended that you avoid using the `setup` function in your custom components. Other constructs available in Simscape language let you achieve the same results without compromising run-time capabilities.

Task	Recommended Technique
Validate parameters	Use an `assert` construct. For more information, see “Programming Run-Time Errors and Warnings” on page 2-48.
Compute derived parameters	Use declaration functions. For more information, see “Declaration Functions” on page 3-22.
Set initial conditions	Assign variable priority and target value. For more information, see “Variable Priority for Model Initialization” on page 2-7.
Designate source for domain parameters	Use direct assignment to a domain parameter in the component node declaration. For more information, see “Source Components” on page 2-98.

See Also

`assert`

Topics

“Programming Run-Time Errors and Warnings” on page 2-48
“Declaration Functions” on page 3-22
“Variable Priority for Model Initialization” on page 2-7
“Source Components” on page 2-98

Introduced in R2008b

tablelookup

Return value based on interpolating set of data points

Syntax

```
tablelookup(x1d, x2d, x3d, x4d, fd, x1, x2, x3, x4, interpolation = linear|
smooth, extrapolation = linear|nearest|error)
```

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, two-dimensional, three-dimensional, and four-dimensional lookup tables. The full syntax is:

```
tablelookup(x1d, x2d, x3d, x4d, fd, x1, x2, x3, x4, interpolation = linear|
smooth, extrapolation = linear|nearest|error)
```

`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 and three-dimensional table lookup.
`x3d`	Data set of input values along the third direction, specified as a one-dimensional array. The values must be strictly monotonic, either increasing or decreasing. This argument is used only for the three-dimensional table lookup.
`x4d`	Data set of input values along the fourth direction, specified as a one-dimensional array. The values must be strictly monotonic, either increasing or decreasing. This argument is used only for the three-dimensional table lookup.

`fd`	Data set of output values for the table lookup. This is a required argument. For one-dimensional table lookup, `fd` must be a one-dimensional array of the same size as `x1d`. For two-dimensional table lookup, `fd` must be a two-dimensional array, 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 `fd` must be an `m`-by-`n` matrix. For three-dimensional table lookup, `fd` must be a three-dimensional array, with the size matching the dimensions defined by the input data sets. For example, if `x1d` is a 1-by-`m` array, `x2d` is a 1-by-`n` array, and `x3d` is a 1-by-`p` array, then `fd` must be an `m`-by-`n`-by-`p` array. For four-dimensional table lookup, `fd` must be a four-dimensional array, with the size matching the dimensions defined by the input data sets. For example, if `x1d` is a 1-by-`m` array, `x2d` is a 1-by-`n` array, `x3d` is a 1-by-`p` array, and `x4d` is a 1-by-`q` array, then `fd` must be an `m`-by-`n`-by-`p`-by-`q` array.
`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 and three-dimensional table lookup.
`x3`	The input value along the third direction. Its units must be commensurate with the units of `x3d`. This argument is used only for the three-dimensional table lookup.
`x4`	The input value along the fourth direction. Its units must be commensurate with the units of `x4d`. This argument is used only for the four-dimensional table lookup.
`interpolation = linear|smooth`	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|error`	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 and three-dimensional table lookup, uses an extension of linear algorithm for multidimensional interpolation, by performing linear interpolation in first direction, then in second direction, and then in third direction. Use this method to get the best performance.

- `smooth` — Uses a modified Akima algorithm. For more information, see `makima`. Use this method to produce a continuous curve or surface with continuous first-order derivatives.

The `extrapolation` argument values are:

- `linear`— Extends from the edge of the interpolation region linearly. The slope of the linear extrapolation is equal to the slope of the interpolated curve or surface at the edge of the interpolation region. Use this method to produce a curve or surface with continuous value and continuous first-order derivatives at the boundary between the interpolation region and the extrapolation region.
- `nearest` — Extends from the edge of the interpolation region as a constant. The value of the nearest extrapolation is equal to the value of the interpolated curve or surface at the edge of the interpolation region. Use this method to produce a curve or surface with continuous value at the boundary between the interpolation region and the extrapolation region that does not go above the highest point in the data or below the lowest point in the data.
- `error` — Generates an error when the input value is outside the range specified in the lookup table.

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

When inputs x1, x2, x3, and x4…	**The `tablelookup` function…**
Match the values in the input data sets, `x1d`, `x2d`, `x3d`, and `x4d`	Outputs the corresponding table value, `fd`
Do not match the values 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 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`, `x3d`, `x4d`, and `fd`:

- For one-dimensional table lookup, `x1d` and `fd` must be one-dimensional arrays of the same size.
- For two-dimensional table lookup, `x1d` and `x2d` must be one-dimensional arrays, and `fd` 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 `fd` must be an `m`-by-`n` matrix.
- For three-dimensional table lookup, `x1d`, `x2d`, and `x3d` must be one-dimensional arrays, and `fd` must be a three-dimensional array, with the size matching the dimensions defined by the input data sets. For example, if `x1d` is a 1-by-`m` array, `x2d` is a 1-by-`n` array, and `x3d` is a 1-by-`p` array, then `fd` must be an `m`-by-`n`-by-`p` array.
- For four-dimensional table lookup, `x1d`, `x2d`, `x3d`, and `x4d` must be one-dimensional arrays, and `fd` must be a four-dimensional array, with the size matching the dimensions defined by the input data sets. For example, if `x1d` is a 1-by-`m` array, `x2d` is a 1-by-`n` array, `x3d` is a 1-by-`p` array, and `x4d` is a 1-by-`q` array, then `fd` must be an `m`-by-`n`-by-`p`-by-`q` array.
- The `x1d`, `x2d`, `x3d`, and `x4d` values must be strictly monotonic, either increasing or decreasing.
- For smooth interpolation, each data set of input values must contain at least three values. For linear interpolation, two values are sufficient.

Using Enumerations for Interpolation and Extrapolation Options

The Foundation library includes built-in enumerations, `interpolation.m` and `extrapolation.m`:

```
classdef interpolation < int32
   enumeration
       linear (1)
       smooth (2)
   end
   methods(Static)
    function map = displayText()
      map = containers.Map;
      map('linear') = 'Linear';
      map('smooth') = 'Smooth';
    end
  end
end
```

```
classdef extrapolation < int32
   enumeration
       linear (1)
       nearest (2)
       error (3)
   end
   methods(Static)
    function map = displayText()
      map = containers.Map;
      map('linear') = 'Linear';
      map('nearest') = 'Nearest';
      map('error') = 'Error';
    end
  end
end
```

These enumerations are located in the directory `*matlabroot*\toolbox\physmod\simscape\library\m\+simscape\+enum`.

You can use these enumerations to declare component parameters, and then use these parameters as `tablelookup` function arguments. For more information, see the “User-Specified Interpolation and Extrapolation Methods” on page 5-75 example and “Using Enumeration in Function Arguments” on page 3-19.

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 2-46.

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
   f = 0;
 end
 parameters (Size=variable)
   x1d = [1 2 3 4];
   x2d = [1 2 3];
   fd = [1 2 3; 3 4 5; 5 6 7; 7 8 9];
 end
 equations
   f == tablelookup(x1d, x2d, fd, u1, u2, interpolation=smooth, extrapolation=nearest);
 end
end
```

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

The interpolation uses the modified Akima algorithm, makima. The extrapolation uses the nearest value of fd for out-of-range u1 and u2 values.

User-Specified Interpolation and Extrapolation Methods

This example is similar to the previous one on page 5-75, but it gives the block user control over the interpolation and extrapolation methods.

```
import simscape.enum.*
component tlu_2d_enum
 inputs
   u1 = 0;
   u2 = 0;
 end
 outputs
```

```
    f = 0;
  end
  parameters (Size=variable)
    x1d = [1 2 3 4];
    x2d = [1 2 3];
    fd = [1 2 3; 3 4 5; 5 6 7; 7 8 9];
  end
  parameters
    interp = interpolation.linear; % Interpolation method
    extrap = extrapolation.linear; % Extrapolation method
  end
  equations
    f == tablelookup(x1d, x2d, fd, u1, u2, interpolation=interp, extrapolation=extrap);
  end
end
```

The component imports the built-in enumerations, and then uses them to declare two additional parameters: `interp` (**Interpolation method**) and `extrap` (**Extrapolation method**). The `tablelookup` function uses these parameters as arguments, to specify the interpolation and extrapolation methods. For more information, see “Using Enumeration in Function Arguments” on page 3-19.

The block generated from this component will have the **Interpolation method** and **Extrapolation method** parameters, both with the default value of `Linear`. The block user can select any other interpolation and extrapolation options.

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 2-46.

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

See Also

PS Lookup Table (1D) | PS Lookup Table (2D) | PS Lookup Table (3D) | PS Lookup Table (4D) | `equations` | `makima`

Topics

"Using Lookup Tables in Equations" on page 2-46

Introduced in R2012a

through

Establish relationship between component variables and nodes

Syntax

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

Description

Note `through` will be removed in a future release. Use `branches` instead. For more information, see “Define Relationship Between Component Variables and Nodes” on page 2-21.

`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`

`branches`

Introduced in R2008b

time

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 component outputs $y = \sin(\omega t)$:

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

See Also

`equations`

Topics

“Use Simulation Time in Equations” on page 2-30

Introduced in R2008b

transitions

Define transitions between modes in mode chart

Syntax

```
transitions from_mode -> to_mode : predicate_condition end
```

Description

`transitions` begins a transitions declaration block in a mode chart. The `transitions` block, terminated by an `end` keyword, can contain one or more transition constructs.

Generally, each transition construct has the following syntax:

```
from_mode -> to_mode : predicate_condition
```

where:

- *`from_mode`* is the mode active before the transition.
- *`to_mode`* is the mode active after the transition.
- *`predicate_condition`* is the expression that needs to be true for the transition to happen.

For example, if a mode chart declares two modes, `m1` and `m2`, the following syntax specifies that the system transitions from mode `m1` to mode `m2` when the `p1` predicate is true:

```
transitions
    m1 -> m2 : p1
end
```

To implement a state reset, mode charts can contain instantaneous modes and compound transitions. Compound transitions have the following syntax:

```
from_mode -> instantaneous_mode -> to_mode : predicate_condition
```

where:

- *`from_mode`* is the mode active before the transition.
- *`instantaneous_mode`* is the mode active for one event iteration during the transition. Only one instantaneous mode is allowed per transition, therefore, a compound transition cannot contain more than three modes.
- *`to_mode`* is the mode active after the transition. For a compound transition, *`to_mode`* can be the same as *`from_mode`*.
- *`predicate_condition`* is the expression that needs to be true for the transition to happen.

For example, in this compound transition, when predicate `t` becomes true, the system transitions from mode `A` to mode `B`, performs one event iteration, and then immediately transitions to mode `C`.

```
A -> B -> C : t
```

Transition Precedence and Execution Rules

If multiple predicates become true simultaneously, the transition priority is defined by the order they are listed. For example, a mode chart declares three modes, `m1`, `m2`, and `m3`, and defines the following transitions:

```
transitions
    m1 -> m2 : p1
    m1 -> m3 : p2
end
```

If predicates `p1` and `p2` become true simultaneously, the system transitions from mode `m1` to mode `m2` (the first transition listed).

At initialization time, the solver sets the initial mode first, and then checks the transitions. If a transition predicate is true at initialization time, the system might start in a different mode than that listed first (or specified by the `initial` construct). For example, consider a mode chart that declares two modes, `m1` and `m2`, and defines the following transition:

```
transitions
    m1 -> m2 : p1
end
```

If predicate `p1` is true at initialization time, the system immediately transitions from mode `m1` (the first mode listed) to mode `m2`, and simulation starts in mode `m2`.

Similarly, if a transition predicate is still true after completing the transition, the system can enter an infinite loop and eventually generate an error. This modeling error is more prevalent with compound transitions, where, after completing the transition, the system often ends up in the same mode where it was before entering the transition. To avoid this situation, try to model compound transitions in such a way that the instantaneous mode invalidates the predicate. For more information, see “State Reset Example” on page 3-11.

See Also

`initial` | `modecharts` | `modes`

Topics

“Mode Chart Modeling” on page 3-2
“Switch with Hysteresis” on page 3-5
“State Reset Modeling” on page 3-11

Introduced in R2017a

value

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 on page 2-5, 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 equation section 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.

See Also

`parameters` | `variables`

Topics

“Declaring a Member as a Value with Unit” on page 2-5

Introduced in R2008b

variables

Define domain or component variables

Syntax

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

```
variables comp_var2 = {value = {value,'unit'}, priority = priority.value, nominal = {value,'unit
```

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

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

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 on page 2-5. `value` is the initial value. `unit` is a valid unit string, defined in the unit registry.

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

For component variables, you can additionally specify the initialization priority, as well as nominal value and unit, by declaring the variable as a field array.

```
variables
    comp_var2 = { value = { value , 'unit' }, priority = priority.value, nominal = { value , 'un
end
```

The first field in the array is `value` (value with unit on page 2-5). The other two fields are optional and can come in any order.

The `priority` field can be one of three values listed in the following table:

Priority field in Simscape language	Resulting default priority in the block dialog box
`priority = priority.high`	`High`
`priority = priority.low`	`Low`
`priority = priority.none` (this is the default)	`None`

Note It is recommended that you use the `priority` attribute sparingly. The default priority value, `priority.none` (which is equivalent to leaving out the `priority` attribute entirely), is suitable in most cases. The block user can modify the variable priority value, as needed, in the **Variables** tab of the block dialog box prior to simulation.

The `nominal` field must be a value with unit on page 2-5, where `value` is the nominal value, that is, the expected magnitude of the variable. `unit` is a valid unit string, defined in the unit registry.

Note It is recommended that you use the `nominal` attribute sparingly. The default nominal values, which come from the model value-unit table, are suitable in most cases. The block user can also modify the nominal values and units for individual blocks by using either the Property Inspector or `set_param` and `get_param` functions, if needed. For more information, see "Modify Nominal Values for a Block Variable".

You can also specify the variable name, the way you want it to appear in the **Variables** tab of the block dialog box, as a comment:

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

In a domain file, the following syntax defines an Across variable, `domain_across1`, as a value with unit on page 2-5. `value` is the initial value. `unit` is a valid unit string, defined in the unit registry.

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

In a domain file, the following syntax defines a Through variable, `domain_through1`, as a value with unit on page 2-5. `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

This example initializes the variable `w` (angular velocity) as 0 rad/s:

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

This example initializes the variable `x` (spring deformation) as 0 mm, with high priority:

```
variables
    x = { value = { 0 , 'mm' }, priority = priority.high }; % Spring deformation
end
```

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

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

See Also

`value`

Topics

“Declare Component Variables” on page 2-7
“Declare Through and Across Variables for a Domain” on page 2-6

Introduced in R2008b

Simscape Foundation Domains

Foundation Domain Types and Directory Structure

Simscape software comes with the following Foundation domains:

- “Electrical Domain” on page 6-4
- “Three-Phase Electrical Domain” on page 6-5
- “Gas Domain” on page 6-6
- “Hydraulic Domain” on page 6-10
- “Isothermal Liquid Domain” on page 6-11
- “Magnetic Domain” on page 6-13
- “Mechanical Rotational Domain” on page 6-14
- “Mechanical Translational Domain” on page 6-15
- “Moist Air Domain” on page 6-16
- “Moist Air Source Domain” on page 6-20
- “Thermal Domain” on page 6-22
- “Thermal Liquid Domain” on page 6-23
- “Two-Phase Fluid Domain” on page 6-26

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
| |-- three_phase.ssc
| |-- ...
|-- +gas
| |-- gas.ssc
| |-- ...
|-- +hydraulic
| |-- hydraulic.ssc
| |-- ...
|-- +isothermal_liquid
| |-- isothermal_liquid.ssc
| |-- ...
|-- +magnetic
| |-- magnetic.ssc
| |-- ...
|-- +mechanical
| |-- +rotational
| | |-- rotational.ssc
| | |-- ...
| |-- +translational
| | |-- translational.ssc
| | |-- ...
|-- +moist_air
| |-- moist_air.ssc
| |-- moist_air_source.ssc
| |-- ...
|-- +pneumatic  (kept for compatibility purposes)
```

```
| |-- pneumatic.ssc
| |-- ...
|-- +thermal
| |-- thermal.ssc
| |-- ...
|-- +thermal_liquid
| |-- thermal_liquid.ssc
| |-- ...
|-- +two_phase_fluid
| |-- two_phase_fluid.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 2005-2013 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
```

Three-Phase Electrical Domain

The three-phase electrical domain declaration is shown below.

```
domain three_phase
    % Three-Phase Electrical Domain

    % Copyright 2012-2013 The MathWorks, Inc.

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

    variables
        V = { [ 0 0 0 ], 'V' };
    end

    variables(Balancing = true)
        I = { [ 0 0 0 ], 'A' };
    end

end
```

It contains the following variables and parameters:

- Across variable *V* (voltage), declared as a three-element row vector, in volts
- Through variable *I* (current), declared as a three-element row vector, 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.three_phase
```

Gas Domain

To view the complete domain source file, at the MATLAB Command prompt, type:

```
open([matlabroot '/toolbox/physmod/simscape/library/m/+foundation/+gas/gas.ssc'])
```

Abbreviated gas domain declaration is shown below, with intermediate lookup table values omitted for readability.

```
domain gas
% Gas Domain

% Copyright 2016 The MathWorks, Inc.

parameters
    gas_spec = {1, '1'}; % Gas specification
    %                      1 - Perfect
    %                      2 - Semiperfect
    %                      3 - Real

    % Perfect gas properties

    R      = {0.287,             'kJ/(kg*K)'}; % Specific gas constant
    Z      = {1,                 '1'        }; % Compressibility factor
    T_ref  = {293.15,            'K'        }; % Reference temperature for gas properties
    h_ref  = {420,               'kJ/kg'    }; % Specific enthalpy at reference temperature
    cp_ref = {1,                 'kJ/(kg*K)'}; % Specific heat at constant pressure
    cv_ref = {0.713,             'kJ/(kg*K)'}; % Specific heat at constant volume
    mu_ref = {18,                'uPa*s'    }; % Dynamic viscosity
    k_ref  = {26,                'mW/(m*K)' }; % Thermal conductivity
    Pr_ref = {0.692307692307692, '1'        }; % Prandtl number

    % Semiperfect gas properties

    T_TLU1 = {[150:10:200, 250:50:1000, 1500, 2000]', 'K'}; % Temperature vector

    log_T_TLU1 = {[
        5.01063529409626
        5.07517381523383
        ...
        7.60090245954208
        ], '1'}; % Log temperature vector

    h_TLU1 = {[
        275.264783730547
        285.377054177734
        ...
        2377.14064127409
        ], 'kJ/kg'}; % Specific enthalpy vector

    cp_TLU1 = {[
        1.01211492398124
        1.01042105529234
        ...
        1.24628356718428
        ], 'kJ/(kg*K)'}; % Specific heat at constant pressure vector

    cv_TLU1 = {[
        0.725174216111164
        0.723480347422265
        ...
        0.959342859314206
        ], 'kJ/(kg*K)'}; % Specific heat at constant volume vector

    mu_TLU1 = {[
        10.3766056544352
        10.9908682444892
        ...
        68.0682900809450
        ], 'uPa*s'}; % Dynamic viscosity vector

    k_TLU1 = {[
        14.1517155766309
        15.0474512994325
        ...
        114.486299090693
        ], 'mW/(m*K)'}; % Thermal conductivity vector
```

```
Pr_TLU1 = {[
    0.742123270231960
    0.738025627675206
    ...
    0.740982912785154
    ], '1'}; % Prandtl number vector

a_TLU1 = {[
    245.095563145758
    253.217606015000
    ...
    863.440849227825
    ], 'm/s'}; % Speed of sound vector

int_dh_T_TLU1 = {[
    0
    0.0652630980004620
    0.126478959779276
    ...
    2.79681971660776
    ], 'kJ/(kg*K)'}; % integral of dh/T vector

% Real gas properties

% Default gas property tables for dry air
% Rows of the tables correspond to the temperature vector
% Columns of the tables correspond to the pressure vector

T_TLU2 = {[150:10:200, 250:50:1000, 1500, 2000]',              'K'  }; % Temperature vector
p_TLU2 = {[0.01:0.01:0.1, 0.12, 0.15, 0.2, 0.5, 1, 2, 5, 10]', 'MPa'}; % Pressure vector

log_T_TLU2 = {[
    5.01063529409626
    5.07517381523383
    ...
    7.60090245954208
    ], '1'}; % Log temperature vector

log_p_TLU2 = {[
    9.21034037197618
    9.90348755253613
    ...
    16.1180956509583
    ], '1'}; % Log pressure vector

log_rho_TLU2 = {[
    -1.45933859209149 -0.765580954956293 ... 2.84006136461620
    ], '1'}; % Log density table

s_TLU2 = {[
    3.85666832168988 3.65733557342939 ... 4.66584072487367
    ], 'kJ/(kg*K)'}; % Specific entropy table

h_TLU2 = {[
    276.007989595737 275.926922934925 ... 2386.79535914098
    ], 'kJ/kg'}; % Specific enthalpy table

cp_TLU2 = {[
    1.00320557010184 1.00416915257750 ... 1.24767439351222
    ], 'kJ/(kg*K)'}; % Specific heat at constant pressure table

cv_TLU2 = {[
    0.715425577953031 0.715655648411093    ... 0.960303115685940
    ], 'kJ/(kg*K)'}; % Specific heat at constant volume table

mu_TLU2 = {[
    10.3604759816291 10.3621937105615 ... 68.3249440282350
    ], 'uPa*s'}; % Dynamic viscosity table

k_TLU2 = {[
    14.0896194596466 14.0962928994967 ... 114.905858092359
    ], 'mW/(m*K)'}; % Thermal conductivity table

Pr_TLU2 = {[
    0.737684026417089 0.738165370950116 ... 0.741888050943969
    ], '1'}; % Prandtl number table

a_TLU2 = {[
    245.567929192228 245.496359667264 ... 878.939999571000
    ], 'm/s'}; % Speed of sound table
    log_drho_dp_TLU2 = {[
        -10.6690690699104 -10.6678475863467 ... -13.2956456388403
        ], '1'}; % Log derivative of density with respect to pressure table

    log_drho_dT_TLU2 = {[
        -6.46809263806814 -5.77245144865022 ... -4.77782516923660
        ], '1'}; % Log derivative of density with respect to temperature table

    drhou_dp_TLU2 = {[
        5.41617782089664 5.42024592837099 ... 3.03095417965253
        ], '1'}; % Derivative of internal energy per unit volume with respect to pressure table

    drhou_dT_TLU2 = {[
        -0.195280173069178 -0.391714814336739 ... 1.27305147462835
        ], 'kJ/(m^3*K)'}; % Derivative of internal energy per unit volume with respect to temperature table

    pT_region_flag   = {1,            '1'}; % Valid pressure-temperature region parameterization
    %                                         1 - Range of gas property tables
    %                                         2 - Specified minimum and maximum values
    %                                         3 - Validity matrix
    pT_validity_TLU2 = {ones(24, 18), '1'}; % Pressure-temperature validity matrix

    T_min = {150,  'K'  }; % Minimum valid temperature
    T_max = {2000, 'K'  }; % Maximum valid temperature
    p_min = {0.01, 'MPa'}; % Minimum valid pressure
    p_max = {10,   'MPa'}; % Maximum valid pressure

    p_atm = {0.101325, 'MPa'}; % Atmospheric pressure

    T_unit       = {1, 'K'          }; % Unit for log temperature
    p_unit       = {1, 'Pa'         }; % Unit for log pressure
    rho_unit     = {1, 'kg/m^3'     }; % Unit for log density
    drho_dp_unit = {1, 'kg/(m^3*Pa)'}; % Unit for log derivative of density with respect to pressure
    drho_dT_unit = {1, 'kg/(m^3*K)' }; % Unit for log derivative of density with respect to temperature

    log_ZR = {5.65948221575962, '1'}; % Log of compressibility factor times specific gas constant

    k_cp = {26e-06, 'kg/(m*s)'}; % Ratio of thermal conductivity to specific heat at typical operating conditions

    max_aspect_ratio = {5, '1'}; % Maximum component aspect ratio (length/diameter) for thermal conduction
end

variables
    p = {0.1, 'MPa'}; % Pressure
    T = {300, 'K'  }; % Temperature
end

variables (Balancing=true)
    mdot = {0, 'kg/s'}; % Mass flow rate
    Phi  = {0, 'kW'  }; % Energy flow rate
end

end
```

The domain declaration contains the following variables and parameters:

- Across variable *p* (absolute pressure), in MPa
- Through variable *mdot* (mass flow rate), in kg/s
- Across variable *T* (temperature), in K
- Through variable *Phi* (energy flow rate), in kW
- Parameter *T_min*, defining the minimum allowable temperature
- Parameter *T_max*, defining the maximum allowable temperature
- Parameter *p_min*, defining the minimum allowable pressure
- Parameter *p_max*, defining the maximum allowable pressure
- Parameter *p_atm*, defining the atmospheric pressure

Parameter *gas_spec* provides a choice of three gas models:

- `1` — Perfect (default)
- `2` — Semiperfect
- `3` — Real

In the Foundation Gas library, the Gas Properties (G) block serves as the source for domain parameter values, including the selection of the gas model. For more information on propagation of domain parameters, see “Working with Domain Parameters” on page 2-98.

The domain declaration also contains sets of parameters that define gas properties for each gas model.

Properties for semiperfect and real gas are in the form of lookup table data. These parameter declarations propagate to the components connected to the Gas domain, and therefore you can use them in the `tablelookup` function in the component equations.

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

```
foundation.gas.gas
```

Hydraulic Domain

Note Isothermal liquid domain, introduced in R2020a, provides a more robust and flexible way of modeling isothermal hydraulic systems.

The hydraulic domain declaration is shown below.

```
domain hydraulic
% Hydraulic Domain

% Copyright 2005-2013 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* (gauge pressure), in Pa
- Through variable *q* (volumetric 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`

Isothermal Liquid Domain

The isothermal liquid domain declaration is shown below.

```
domain isothermal_liquid
% Isothermal Liquid Domain

% Copyright 2019 The MathWorks, Inc.

parameters
    rho_L_atm           = { 998.21,    'kg/m^3' }; % Liquid density at atmospheric pressure (no entrained air)
    entrained_air_model = foundation.enum.entrained_air_model.const; % Entrained air model
    %                                                                       1 - const
    %                                                                       2 - linear
    bulk_modulus_model = foundation.enum.bulk_modulus_model.const; % Isothermal bulk modulus model
    %                                               1 - const
    %                                               2 - linear
    beta_gain          = { 6,          '1'      }; % Isothermal bulk modulus vs. pressure increase gain
    beta_L_atm         = { 2.1791e9,   'Pa'     }; % Liquid isothermal bulk modulus at atmospheric pressure (no entrained air)
    nu_atm             = { 1.0034e-6,  'm^2/s'  }; % Kinematic viscosity at atmospheric pressure
    air_ratio          = { 0 ,         '1'      }; % Entrained air-to-liquid volumetric ratio at atmospheric pressure
    polytropic_index   = { 1.0,        '1'      }; % Air polytropic index
    rho_g_atm          = {1.225,       'kg/m^3' }; % Gas (air) density at atmospheric condition
    p_min              = { 0.1,        'Pa'     }; % Minimum valid pressure
    p_atm              = { 0.101325,   'MPa'    }; % Atmospheric pressure
    p_crit             = { 3,          'MPa'    }; % Pressure at which all entrained air is dissolved
end

variables
    p = { 0.1, 'MPa' }; % Pressure
end

variables(Balancing = true)
    mdot = { 0 , 'kg/s' }; % Mass flow rate
end

end
```

It contains the following variables and parameters:

- Across variable *p* (absolute pressure), in MPa
- Through variable *mdot* (mass flow rate), in kg/s
- Parameter *rho_L_atm*, defining the liquid density at atmospheric pressure, with zero entrained air
- Enumerated parameter *entrained_air_model*, defining the entrained air parametrization, with two values:
  - 0 — Entrained air is constant
  - 1 — Entrained air is a function of pressure
- Enumerated parameter *bulk_modulus_model*, defining the bulk modulus parametrization, with two values:
  - 0 — Bulk modulus is constant
  - 1 — Bulk modulus is a function of pressure
- Parameter *beta_gain*, defining the ratio of bulk modulus to pressure increase, for when the bulk modulus is a function of pressure
- Parameter *beta_L_atm*, defining the liquid isothermal bulk modulus at atmospheric pressure, with zero entrained air
- Parameter *nu_atm*, defining the kinematic viscosity at atmospheric pressure
- Parameter *air_ratio*, defining the entrained air-to-liquid volumetric ratio at atmospheric pressure
- Parameter *polytropic_index*, defining the air polytropic index

- Parameter *rho_g_atm*, defining the air density at atmospheric condition
- Parameter *p_min*, defining the minimum valid pressure
- Parameter *p_atm*, defining the atmospheric pressure
- Parameter *p_crit*, defining the pressure at which all entrained air is dissolved, for when the amount of entrained air is a function of pressure

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

```
foundation.isothermal_liquid.isothermal_liquid
```

Magnetic Domain

The magnetic domain declaration is shown below.

```
domain magnetic
% Magnetic Domain

% Copyright 2009-2013 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 2005-2013 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 2005-2013 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
```

Moist Air Domain

To view the complete domain source file, at the MATLAB Command prompt, type:

```
open([matlabroot '/toolbox/physmod/simscape/library/m/+foundation/+moist_air/moist_air.ssc'])
```

Abbreviated moist air domain declaration is shown below, with intermediate lookup table values omitted for readability.

```
domain moist_air
% Moist Air Domain

% Copyright 2017 The MathWorks, Inc.

parameters
    trace_gas_model = foundation.enum.trace_gas_model.none; % Trace gas model
    %                                                         1 - none
    %                                                         2 - track_fraction
    %                                                         3 - track_properties

    R_a = {287.047, 'J/(kg*K)'}; % Dry air specific gas constant
    R_w = {461.523, 'J/(kg*K)'}; % Water vapor specific gas constant
    R_g = {188.923, 'J/(kg*K)'}; % Trace gas specific gas constant

    T_TLU = {[-56.55, -50:10:-10, -5:1:5, 10:10:350]', 'degC'}; % Temperature vector

    log_p_ws_TLU = [
        0.537480914463376
        1.37059832527040
        ...
        16.4965444877527
        16.6206369090880]; % Log of water vapor saturation pressure vector in Pa

    h_w_vap_TLU = {[
        2836.88241275372
        2837.81392500514
        ...
        1027.62017777647
        892.733785613825], 'kJ/kg'}; % Water specific enthalpy of vaporization vector

    h_a_TLU = {[
        342.416126230579
        349.005511058471
        ...
        747.258774447567
        757.813011774199], 'kJ/kg'}; % Dry air specific enthalpy vector

    h_w_TLU = {[
        2396.55944251649
        2408.68643343608
        ...
        3155.43043805905
        3175.80160435813], 'kJ/kg'}; % Water vapor specific enthalpy vector

    h_g_TLU = {[
        342.416126230579
        349.005511058471
        ...
        747.258774447567
        757.813011774199], 'kJ/kg'}; % Trace gas specific enthalpy vector

    mu_a_TLU = {[
        14.2568883320012
        14.6140127728333
        ...
        31.2307628592324
        31.5791070262086], 'uPa*s'}; % Dry air dynamic viscosity vector

    mu_w_TLU = {[
        6.81365662228272
        7.04953750742707
        ...
        21.1317199525111
        21.4937680016671], 'uPa*s'}; % Water vapor dynamic viscosity vector

    mu_g_TLU = {[
        14.2568883320012
        14.6140127728333
        ...
        31.2307628592324
        31.5791070262086], 'uPa*s'}; % Trace gas dynamic viscosity vector

    k_a_TLU = {[
        19.8808489374933
        20.4162454629695
        ...
        46.7832370779530
        47.3667074066625], 'mW/(m*K)'}; % Dry air thermal conductivity vector

    k_w_TLU = {[
        11.4628821597600
        11.9419974889350
        ...
        43.1675775109350
        44.0380174089350], 'mW/(m*K)'}; % Water vapor thermal conductivity vector

    k_g_TLU = {[
        19.8808489374933
        20.4162454629695
        ...
        46.7832370779530
        47.3667074066625], 'mW/(m*K)'}; % Trace gas thermal conductivity vector

    cp_a_coeff = {[
        1.02664779928781
        -0.000177515573577911
        3.66581785159269e-07], 'kJ/(kg*K)'}; % Dry air specific heat polynomial coefficients

    cp_w_coeff = {[
        1.47965047747103
        0.00120021143370507
        -3.86145131678391e-07], 'kJ/(kg*K)'}; % Water vapor specific heat polynomial coefficients

    cp_g_coeff = {[
        1.02664779928781
        -0.000177515573577911
        3.66581785159269e-07], 'kJ/(kg*K)'}; % Trace gas specific heat polynomial coefficients

    Pr_a_TLU = [
        0.720986465349271
        0.719589372441350
        ...
        0.704694042255749
        0.705614770118245]; % Dry air Prandtl number pressure vector

    Pr_w_TLU = [
        1.02327757654022
        ...
        1.01351190334830
        1.01402827396757]; % Water vapor Prandtl number pressure vector

    Pr_g_TLU = [
        0.720986465349271
        0.719589372441350
        ...
        0.704694042255749
        0.705614770118245]; % Trace gas Prandtl number pressure vector

    int_dh_T_a_TLU = {[
        0
        0.0299709934765051
        ...
```

```
            1.05826245662507
            1.07533673877425], 'kJ/(kg*K)'}; % Dry air integral of dh/T vector

        int_dh_T_w_TLU = {[
            0
            0.0551581028022933
            ...
            1.96804836665268
            2.00100413885432], 'kJ/(kg*K)'}; % Water vapor integral of dh/T vector

        int_dh_T_g_TLU = {[
            0
            0.0299709934765051
            ...
            1.05826245662507
            1.07533673877425], 'kJ/(kg*K)'}; % Trace gas integral of dh/T vector

        D_w = {25, 'mm^2/s'}; % Water vapor diffusivity in air
        D_g = {1,  'mm^2/s'}; % Trace gas diffusivity in air

        p_min = {1,      'kPa' }; % Minimum valid pressure
        p_max = {inf,    'MPa' }; % Maximum valid pressure
        T_min = {-56.55, 'degC'}; % Minimum valid temperature
        T_max = {350,    'degC'}; % Maximum valid temperature

        p_atm = {0.101325, 'MPa' }; % Atmospheric pressure
        T_atm = {20,       'degC'}; % Atmospheric temperature

        rho_a_atm = {1.20412924943656, 'kg/m^3'   }; % Dry air density at reference condition
        cp_a_atm  = {1.00611201935459, 'kJ/(kg*K)'}; % Dry air specific heat at reference condition
        k_a_atm   = {25.8738283029331, 'mW/(m*K)' }; % Dry air thermal conductivity at reference condition
    end

    variables
        p   = {0.1, 'MPa'}; % Pressure
        T   = {300, 'K'  }; % Temperature
        x_w = 0;            % Specific humidity
        x_g = 0;            % Trace gas mass fraction
    end

    variables (Balancing=true)
        mdot   = {0, 'kg/s'}; % Mixture mass flow rate
        Phi    = {0, 'kW'  }; % Mixture energy flow rate
        mdot_w = {0, 'kg/s'}; % Water vapor mass flow rate
        mdot_g = {0, 'kg/s'}; % Trace gas mass flow rate
    end

    end
```

The domain declaration contains the following variables and parameters:

- Across variable *p* (absolute pressure), in MPa
- Through variable *mdot* (mixture mass flow rate), in kg/s
- Across variable *T* (temperature), in K
- Through variable *Phi* (mixture energy flow rate), in kW
- Across variable *x_w* (specific humidity), unitless
- Through variable *mdot_w* (water vapor mass flow rate), in kg/s
- Across variable *x_g* (trace gas mass fraction), unitless
- Through variable *mdot_g* (trace gas mass flow rate), in kg/s
- Parameter *p_min*, defining the minimum allowable pressure
- Parameter *p_max*, defining the maximum allowable pressure
- Parameter *T_min*, defining the minimum allowable temperature

- Parameter *T_max*, defining the maximum allowable temperature
- Parameter *p_atm*, defining the atmospheric pressure
- Parameter *T_atm*, defining the atmospheric temperature

Parameter *trace_gas_model* provides a choice of three trace gas models:

- `foundation.enum.trace_gas_model.none` — None
- `foundation.enum.trace_gas_model.track_fraction` — Track mass fraction only
- `foundation.enum.trace_gas_model.track_properties` — Track mass fraction and gas properties

In the Foundation Moist Air library, the Moist Air Properties (MA) block serves as the source for domain parameter values, including the selection of the trace gas model. For more information on propagation of domain parameters, see "Working with Domain Parameters" on page 2-98.

The moist air mixture is composed of three gas species. The default domain parameter values correspond to dry air, water vapor, and carbon dioxide:

- R_a = {287.047, 'J/(kg*K)'}; % Dry air specific gas constant
- R_w = {461.523, 'J/(kg*K)'}; % Water vapor specific gas constant
- R_g = {188.923, 'J/(kg*K)'}; % Trace gas specific gas constant

You can modify these parameter values in the Moist Air Properties (MA) block to model any three-species gas mixture.

The domain declaration also contains sets of parameters that define various dry air, water vapor, and trace gas properties in the form of lookup table data. The table lookup is with respect to the temperature vector, *T_TLU*. These parameter declarations propagate to the components connected to the Moist Air domain, and therefore you can use them in the `tablelookup` function in the component equations.

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

```
foundation.moist_air.moist_air
```

See Also

"Moist Air Source Domain" on page 6-20

More About

- "Modeling Moist Air Systems"
- "Working with Domain Parameters" on page 2-98

Moist Air Source Domain

This domain is used only for connecting sources of moisture and trace gas to components with internal moist air volume.

To view the complete domain source file, at the MATLAB Command prompt, type:

```
open([matlabroot '/toolbox/physmod/simscape/library/m/+foundation/+moist_air/moist_air_source.ssc
```

Abbreviated moist air source domain declaration is shown below, with intermediate lookup table values omitted for readability.

```
domain moist_air_source
% Moist Air Source Domain
% This domain is used only for connecting sources of moisture and trace gas
% to moist air components.

% Copyright 2017 The MathWorks, Inc.

parameters
    trace_gas_model = foundation.enum.trace_gas_model.track_properties; % Trace gas model
    %                                                                     1 - none
    %                                                                     2 - track_fraction
    %                                                                     3 - track_properties

    T_TLU = {[-56.55, -50:10:-10, -5:1:5, 10:10:350]', 'degC'}; % Temperature vector

    h_w_vap_TLU = {[
        2836.88241275372
        2837.81392500514
        ...
        1027.62017777647
        892.733785613825], 'kJ/kg'}; % Water specific enthalpy of vaporization vector

    h_w_TLU = {[
        2396.55944251649
        2408.68643343608
        ...
        3155.43043805905
        3175.80160435813], 'kJ/kg'}; % Water vapor specific enthalpy vector

    h_g_TLU = {[
        439.555216260064
        444.670268200251
        ...
        814.123440770426
        824.984623198037], 'kJ/kg'}; % Trace gas specific enthalpy vector

    T_min = {-56.55, 'degC'}; % Minimum valid temperature
    T_max = {350,    'degC'}; % Maximum valid temperature
    T_atm = {20,     'degC'}; % Atmospheric temperature
end

variables
    T   = {300, 'K'}; % Temperature
    x_w = 0;          % Specific humidity
    x_g = 0;          % Trace gas mass fraction
end

variables (Balancing=true)
    Phi    = {0, 'kW'  }; % Mixture energy flow rate
    mdot_w = {0, 'kg/s'}; % Water vapor mass flow rate
    mdot_g = {0, 'kg/s'}; % Trace gas mass flow rate
end

end
```

The domain declaration contains the following variables and parameters:

- Across variable *T* (temperature), in K
- Through variable *Phi* (mixture energy flow rate), in kW
- Across variable *x_w* (specific humidity), unitless
- Through variable *mdot_w* (water vapor mass flow rate), in kg/s
- Across variable *x_g* (trace gas mass fraction), unitless
- Through variable *mdot_g* (trace gas mass flow rate), in kg/s
- Parameter *T_min*, defining the minimum allowable temperature
- Parameter *T_max*, defining the maximum allowable temperature
- Parameter *T_atm*, defining the atmospheric temperature

Parameter *trace_gas_model* provides a choice of three trace gas models:

- `foundation.enum.trace_gas_model.none` — None
- `foundation.enum.trace_gas_model.track_fraction` — Track mass fraction only
- `foundation.enum.trace_gas_model.track_properties` — Track mass fraction and gas properties

In the Foundation Moist Air library, the Moist Air Properties (MA) block serves as the source for domain parameter values, including the selection of the trace gas model. For more information on propagation of domain parameters, see “Working with Domain Parameters” on page 2-98.

The domain declaration also contains sets of parameters that define water vapor and trace gas properties in the form of lookup table data. The table lookup is with respect to the temperature vector, *T_TLU*. These parameter declarations propagate to the components connected to the Moist Air Source domain, and therefore you can use them in the `tablelookup` function in the component equations.

You do not need to independently specify the water vapor and trace gas properties for the Moist Air Source domain. The Moist Air library blocks with an **S** port are set up in such a way that they propagate the properties from the regular Moist Air domain to the Moist Air Source domain connected to their **S** port. This way, the water vapor and trace gas properties are consistent between the Moist Air domain and the Moist Air Source domain.

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

```
foundation.moist_air.moist_air_source
```

See Also

“Moist Air Source Domain” on page 6-20

More About

- “Modeling Moisture and Trace Gas Levels”
- “Working with Domain Parameters” on page 2-98

Thermal Domain

The thermal domain declaration is shown below.

```
domain thermal
% Thermal domain

% Copyright 2005-2013 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
```

Thermal Liquid Domain

To view the complete domain source file, at the MATLAB Command prompt, type:

```
open([matlabroot '/toolbox/physmod/simscape/library/m/+foundation/+thermal_liquid/thermal_liquid
```

Abbreviated thermal liquid domain declaration is shown below, with intermediate lookup table values omitted for readability.

```
domain thermal_liquid
% Thermal Liquid Domain

% Copyright 2012-2016 The MathWorks, Inc.

parameters (Size=variable)
    % Default liquid property tables for water
    % Rows of the tables correspond to the temperature vector
    % Columns of the tables correspond to the pressure vector

    T_TLU = {[273.1600:10:373.16]', 'K'  }; % Temperature vector
    p_TLU = {[0.01, 0.1, 5:5:50],   'MPa'}; % Pressure vector

    pT_validity_TLU = {[
        1     1     1     1     1     1     1     1     1     1     1     1
        1     1     1     1     1     1     1     1     1     1     1     1
        1     1     1     1     1     1     1     1     1     1     1     1
        1     1     1     1     1     1     1     1     1     1     1     1
        1     1     1     1     1     1     1     1     1     1     1     1
        0     1     1     1     1     1     1     1     1     1     1     1
        0     1     1     1     1     1     1     1     1     1     1     1
        0     1     1     1     1     1     1     1     1     1     1     1
        0     1     1     1     1     1     1     1     1     1     1     1
        0     1     1     1     1     1     1     1     1     1     1     1
        0     1     1     1     1     1     1     1     1     1     1     1
        ], '1'}; % Pressure-temperature validity matrix

    rho_TLU = {[
        999.8    999.8    ...     978.2     980.3
        ], 'kg/m^3'}; % Density table

    u_TLU = {[
        0.0002    0.0018  ...     407.1700  405.9800
        ], 'kJ/kg'}; % Specific internal energy table

    nu_TLU = {[
        1.7917    1.7914  ...    0.3000    0.3007
        ], 'mm^2/s'}; % Kinematic viscosity table

    cp_TLU  = {[
        4.2199    4.2194  ...    4.1245    4.1157
        ], 'kJ/(kg*K)'}; % Specific heat at constant pressure table

    k_TLU = {[
        561.0400  561.0900 ...    703.3500  706.0000
        ], 'mW/(m*K)'}; % Thermal conductivity table

    beta_TLU = {[
        1.9649    1.9654  ...    2.3455    2.3788
        ], 'GPa'}; % Isothermal bulk modulus table

    alpha_TLU = {1e-4 * [
        -0.6790   -0.6760 ...    6.8590    6.8000
        ], '1/K'}; % Isobaric thermal expansion coefficient table

    mu_TLU = {[
        1.79134166000000  ...    0.294776210000000
        ], 'cP'}; % Dynamic viscosity table

    Pr_TLU = {[
        13.4736964762477  ...    1.71842839588810
        ], '1'}; % Prandtl number table
end

parameters
    pT_region_flag   = {1,        '1'       }; % Valid pressure-temperature region parameterization
    %                                            0 - By minimum and maximum value
    %                                            1 - By validity matrix
    T_min            = {273.16,   'K'       }; % Minimum valid temperature
    T_max            = {373.16,   'K'       }; % Maximum valid temperature
    p_min            = {0.01,     'MPa'     }; % Minimum valid pressure
    p_max            = {50,       'MPa'     }; % Maximum valid pressure
    p_atm            = {0.101325, 'MPa'     }; % Atmospheric pressure
    k_cv             = {1.43e-4,  'kg/(m*s)'}; % Ratio of thermal conductivity to specific heat
    max_aspect_ratio = {5,        '1'       }; % Maximum component aspect ratio (length/diameter) for thermal conduction
end

variables
    p = {0.1, 'MPa'}; % Pressure
    T = {300, 'K'  }; % Temperature
end

variables (Balancing=true)
    mdot = {0, 'kg/s'}; % Mass flow rate
    Phi  = {0, 'kW'  }; % Energy flow rate
end

end
```

It contains the following variables and parameters:

- Across variable *p* (absolute pressure), in MPa
- Through variable *mdot* (mass flow rate), in kg/s
- Across variable *T* (temperature), in kelvin
- Through variable *Phi* (energy flow rate), in kW
- Parameter *pT_region_flag*, defining the valid pressure-temperature region parametrization, with two values:
  - 0 — By minimum and maximum value
  - 1 — By validity matrix
- Parameter *T_min*, defining the minimum valid temperature
- Parameter *p_min*, defining the minimum valid pressure
- Parameter *T_max*, defining the maximum valid temperature
- Parameter *p_max*, defining the maximum valid pressure
- Parameter *p_atm*, defining the atmospheric pressure
- Parameter *k_cv*, defining the ratio of thermal conductivity to specific heat
- Parameter *max_aspect_ratio*, defining the maximum component aspect ratio (length/diameter) for thermal conduction

It also contains lookup tables, declared as variable-sized domain parameters, for the following liquid thermodynamic properties:

- Density
- Specific internal energy
- Kinematic viscosity
- Specific heat at constant pressure
- Thermal conductivity
- Isothermal bulk modulus
- Isobaric thermal expansion coefficient
- Dynamic viscosity

- Prandtl number

These variable-sized parameter declarations propagate to the components connected to the Thermal Liquid domain, and therefore you can use them in the `tablelookup` function in the component equations. In particular, the thermal liquid blocks in the Foundation library use these lookup tables for interpolation purposes.

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

```
foundation.thermal_liquid.thermal_liquid
```

Two-Phase Fluid Domain

To view the complete domain source file, at the MATLAB Command prompt, type:

```
open([matlabroot '/toolbox/physmod/simscape/library/m/+foundation/+two_phase_fluid/two_phase_flu
```

Abbreviated two-phase fluid domain declaration is shown below, with intermediate lookup table values omitted for readability.

```
domain two_phase_fluid
% Two-Phase Fluid Domain

% Copyright 2013-2015 The MathWorks, Inc.

parameters
    p_min       = { 0.01,   'MPa'   }; % Minimum valid pressure
    p_max       = { 10,     'MPa'   }; % Maximum valid pressure
    u_min       = { 83,     'kJ/kg' }; % Minimum valid specific internal energy
    u_max       = { 3000,   'kJ/kg' }; % Maximum valid specific internal energy
    p_atm       = { 1,      'atm'   }; % Atmospheric pressure
    G_min        = { 1e-4,   'kg/s'  }; % Minimum thermal conductance coefficient (in terms of specific internal energy)
end

parameters (Size = variable)
    % Default lookup tables as a function of pressure and normalized
    % specific internal energy. Default values are given for water.

    unorm_TLU = {[
        -1
        -0.965517241379310
        ...
        1.965517241379310
        2] , '1' };  % Normalized specific internal energy vector

    p_TLU = {[
        0.0100000000000000
        0.0110069417125221
        ...
        9.08517575651687
        10 ]', 'MPa' }; % Pressure vector

    v_TLU = {[
        0.00100179322007424    0.00100179275967088    ...     0.0346304234950199    0.0314287403997878
        ], 'm^3/kg' }; % Specific volume table

    T_TLU = {[
        292.932206861359    292.932221312314    ...    745.161185789778    749.863646934846
        ], 'K' };    % Temperature table

    nu_TLU = {[
        1.00879736586632    1.00879622386086    ...    1.84426245592388    1.86263886789473
        ], 'J/(g*K)' };   % Specific heat at constant volume table

    k_TLU = {[
        0.597572631285823    0.597573251441156    ...    0.0722108618593745    0.0737522416007857
        ], 'W/(m*K)' }; % Thermal conductivity table

    Pr_TLU = {[
        7.05143974468479    7.05142233958132    ...    0.996712568607034    1.00323505646151
        ], '1' };  % Prandtl number table

    u_liq = {[
        191.795842042090
        199.694279536627
        ...
        1354.80955706624
        1393.53799592228]', 'kJ/kg' };    % Saturated liquid specific internal energy vector

    u_vap = {[
        2437.15737300173
        2439.67287156956
        ...
        2557.44803624027
        2545.19234394635]', 'kJ/kg' };    % Saturated vapor specific internal energy vector
end

variables
```

```
    p = { 0.101325,        'MPa'   };  % Pressure
    u = { 83.905793864039, 'kJ/kg' };  % Specific internal energy
end

variables(Balancing = true)
    mdot = { 0, 'kg/s'    };    % Mass flow rate
    Phi  = { 0, 'kW'    };  % Heat flow rate
end

end
```

The domain declaration contains the following variables and parameters:

- Across variable *p* (absolute pressure), in MPa
- Through variable *mdot* (mass flow rate), in kg/s
- Across variable *u* (specific internal energy), in kJ/kg
- Through variable *Phi* (heat flow rate), in kW
- Parameter *p_min*, defining the minimum allowable pressure
- Parameter *p_max*, defining the maximum allowable pressure
- Parameter *u_min*, defining the minimum allowable specific internal energy
- Parameter *u_max*, defining the maximum allowable specific internal energy
- Parameter *p_atm*, defining the atmospheric pressure
- Parameter *G_min*, defining the minimum thermal conductance coefficient, in terms of specific internal energy

It also contains lookup table data, declared as variable-sized domain parameters, for the following fluid properties:

- Normalized specific internal energy vector
- Pressure vector
- Specific volume table
- Temperature table
- Specific heat at constant volume table
- Thermal conductivity table
- Prandtl number table
- Saturated liquid specific internal energy vector
- Saturated vapor specific internal energy vector

These variable-sized parameter declarations propagate to the components connected to the Two-Phase Fluid domain, and therefore you can use them in the `tablelookup` function in the component equations. In particular, the two-phase fluid blocks in the Foundation library use these lookup tables for interpolation purposes.

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

```
foundation.two_phase_fluid.two_phase_fluid
```

Pneumatic Domain

Note As of R2016b, the gas domain on page 6-6 replaces the pneumatic domain as the recommended way of modeling pneumatic systems. The pneumatic domain definition is still provided with the software, for compatibility reasons. However, it can be removed in a future release.

The pneumatic domain declaration is shown below.

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

% Copyright 2008-2013 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* (absolute 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
```