[Simple First Order System](http://book.xogeny.com/behavior/equations/first_order/)

[An Electrical Example](http://book.xogeny.com/behavior/equations/electrical/)



[http://book.xogeny.com/_static/images/TitleHeading.png](http://book.xogeny.com/)

Release: v0.3.6-1-gc28d7f1-Early Access

[Basic Equations](http://book.xogeny.com/behavior/equations/)

**Getting Physical**

Table Of Contents

* [Basic Equations](http://book.xogeny.com/behavior/equations/)
  + [Getting Physical](http://book.xogeny.com/behavior/equations/physical/)
    - [Physical Units](http://book.xogeny.com/behavior/equations/physical/#physical-units)
    - [Physical Types](http://book.xogeny.com/behavior/equations/physical/#physical-types)

Getting Physical

Although the previous section got us started with representing mathematical behavior, it doesn’t convey any connection to *physical* behavior. In this section, we’ll explore how to build models that represent the modeling of physical behavior. Along the way, we will highlight some of the language features we can leverage that will not only tie these models to physical and engineering domains, but, as we shall see, they can even help us avoid mistakes.

Let’s start with the following example:

**model** **NewtonCooling** "An example of Newton's law of cooling"

**parameter** Real T\_inf "Ambient temperature";

**parameter** Real T0 "Initial temperature";

**parameter** Real h "Convective cooling coefficient";

**parameter** Real A "Surface area";

**parameter** Real m "Mass of thermal capacitance";

**parameter** Real c\_p "Specific heat";

Real T "Temperature";

**initial equation**

T = T0 "Specify initial value for T";

**equation**

m\*c\_p\*der(T) = h\*A\*(T\_inf-T) "Newton's law of cooling";

**end** **NewtonCooling**;

As we saw in the examples in our discussion of *[Simple First Order System](http://book.xogeny.com/behavior/equations/first_order/" \l "first-order)*, the previous example consists of a model definition that includes variables and equations.

However, this time we see the word parameter for the first time. Generally speaking, the parameter keyword is used to indicate variables whose value is known *a priori* (*i.e.*, prior to the simulation). More precisely, parameter is a keyword that specifies the *variability* of a variable. This will be discussed more thoroughly in the section on *[Variability](http://book.xogeny.com/behavior/equations/variables/" \l "variability)*. But for now, we can think of a parameter as a variable whose value we must provide.

Looking at our NewtonCooling example, we see there are five parameters: T\_inf, T0, h, A, m and c\_p. We don’t need to bother explaining what these variables are because the model itself includes a descriptive string for each one. At the moment, there are no values for these parameters, but we will return to that topic shortly. As with all the variables we have seen so far, these are all of type Real.

Let’s examine the rest of this model. The next variable is T (also a Real). Since this variable doesn’t have the parameter qualifier, its value is determined by the equations in the model.

Next we see the two equation sections. The first is an initial equation section which specifies how the variable T should be initialized. It should be pretty clear that the initial value for T is going to be whatever value was given (by us) for the parameter T0.

The other equation is the differential equation that governs the behavior of T. Mathematically, we could express this equation as:

mcpT˙=hA(T∞−T)

but in Modelica, we write it as:

m\*c\_p\*der(T) = h\*A\*(T\_inf-T)

Note that this is really no different from the equation we saw in our FirstOrder model from the [*Simple First Order System*](http://book.xogeny.com/behavior/equations/first_order/#first-order) example.

One thing worth noting is that the equation in our NewtonCooling example contains an **expression** on the left hand side. In Modelica, it is **not** necessary for each equation to be an explicit equation for a single variable. An equation can contain arbitrary expressions on either side of the equals sign. It is the compiler’s job to determine how to use these equations to solve for the variables contained in the equations.

Another thing that distinguishes our NewtonCooling example from the FirstOrder model is that we can independently adjust the different parameter values. Furthermore, these parameter values are tied to physical, measurable properties of the materials or environmental conditions. In other words, this version is slightly more physical than the simple mathematical relationship used in the FirstOrder model because it is related to physical properties.

Now, we can’t really run the NewtonCooling model as is because it lacks *values* for the six parameters. In order to create a model that is ready to be simulated, we need to provide those values, *e.g.*,

**model** **NewtonCoolingWithDefaults** "Cooling example with default parameter values"

**parameter** Real T\_inf=25 "Ambient temperature";

**parameter** Real T0=90 "Initial temperature";

**parameter** Real h=0.7 "Convective cooling coefficient";

**parameter** Real A=1.0 "Surface area";

**parameter** Real m=0.1 "Mass of thermal capacitance";

**parameter** Real c\_p=1.2 "Specific heat";

Real T "Temperature";

**initial equation**

T = T0 "Specify initial value for T";

**equation**

m\*c\_p\*der(T) = h\*A\*(T\_inf-T) "Newton's law of cooling";

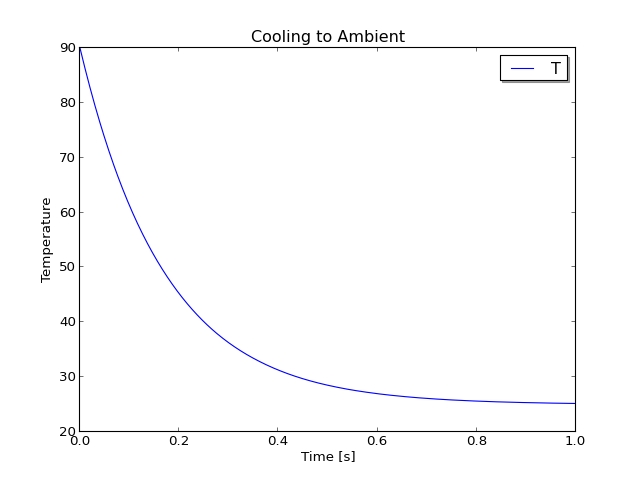
**end** **NewtonCoolingWithDefaults**;

The only real difference here is that each of the parameter variables now has a value specified. One way to think about the NewtonCooling model is that we could not simulate it because it had 7 variables (total) and only one equation (see the section on *[Initialization](http://book.xogeny.com/behavior/equations/initialization/" \l "initialization)* for an explanation of why the initialequation doesn’t really count). However, the NewtonCoolingWithDefaults model has, conceptually speaking, 7 equations (6 of them coming from specifying the values of the parameter variables + one in the equation section) and 7 unknowns.

If we simulate the NewtonCoolingWithDefaults model, we get the following solution for T.

([Source code](http://book.xogeny.com/plots/NCWD.py))

 Simulate NewtonCoolingWithDefaults in your browser



Physical Units

As mentioned already in this section, these examples are a bit more physical because they include individual physical parameters that correspond to individual properties of our real world system. However, we are still missing something. Although these variables represent physical quantities like temperature, mass, *etc.*, we haven’t explicitly given them any physical types.

As you may have already guessed, the variable T is a temperature. This is made clear in the descriptive text associated with the variable. Furthermore, it doesn’t take a very deep analysis of our previous model to determine that T0 and T\_inf must also be temperatures.

But what about the other variables like h or A? What do they represent? Even more important, are the equations **physically consistent**? By physically consistent, we mean that both sides of the equations have the same physical units (*e.g.*, temperature, mass, power).

We could convey the physical units of the different variables more rigorously by actually including them in the variable declarations, like so:

**model** **NewtonCoolingWithUnits** "Cooling example with physical units"

**parameter** Real T\_inf(unit="K")=298.15 "Ambient temperature";

**parameter** Real T0(unit="K")=363.15 "Initial temperature";

**parameter** Real h(unit="W/(m2.K)")=0.7 "Convective cooling coefficient";

**parameter** Real A(unit="m2")=1.0 "Surface area";

**parameter** Real m(unit="kg")=0.1 "Mass of thermal capacitance";

**parameter** Real c\_p(unit="J/(K.kg)")=1.2 "Specific heat";

Real T(unit="K") "Temperature";

**initial equation**

T = T0 "Specify initial value for T";

**equation**

m\*c\_p\*der(T) = h\*A\*(T\_inf-T) "Newton's law of cooling";

**end** **NewtonCoolingWithUnits**;

Note that each of the variable declarations now includes the text (unit="...") to associate a physical unit with the variable. What this additional text does is specify a value for the unit attribute associated with the variable. Attributes are special properties that each variable has. The set of attributes a variable can have depends on the type of the variable (this is discussed in more detail in the upcoming section on [*Variables*](http://book.xogeny.com/behavior/equations/variables/#variables)).

At first glance, it may not seem obvious why specifying the unit attribute (*e.g.*, (unit="K")) is any better than simply adding "Temperature" to the descriptive string following the variable. In fact, one might even argue it is worse because “Temperature” is more descriptive than just a single letter like “K”.

However, setting the unit attribute is actually a more formal approach for two reasons. The first reason is that the Modelica specification defines relationships for all the standard SI unit attributes (*e.g.*, K, kg, m). This includes complex unit types that can be composed of other base units (*e.g.*, N).

The other reason is that the Modelica specification also defines rules for how to compute the units of complex mathematical expressions. In this way, the Modelica specification defines everything that is necessary to **unit check** Modelica models for errors or physical inconsistencies. This is a big win for model developers because adding units not only makes the models clearer, it provides better diagnostics in the case of errors.

Physical Types

But truth be told, there is one drawback of the code for our NewtonCoolingWithUnits example and that is that we have to repeat the unit attribute specification for every variable. Furthermore, as mentioned previously, K isn’t nearly as descriptive as “Temperature”.

Fortunately, we have a simple solution to both problems because Modelica allows us to define *derived types*. So far, all the variables we have declared have been of type Real. The problem with Real is that it could be anything (*e.g.*, a voltage, a current, a temperature). What we’d like to do is narrow things down a bit. This is where derived types come in. To see how to define derived types and then use them in declarations, consider the following example:

**model** **NewtonCoolingWithTypes** "Cooling example with physical types"

*// Types*

**type** **Temperature**=Real(unit="K", min=0);

**type** **ConvectionCoefficient**=Real(unit="W/(m2.K)", min=0);

**type** **Area**=Real(unit="m2", min=0);

**type** **Mass**=Real(unit="kg", min=0);

**type** **SpecificHeat**=Real(unit="J/(K.kg)", min=0);

*// Parameters*

**parameter** Temperature T\_inf=298.15 "Ambient temperature";

**parameter** Temperature T0=363.15 "Initial temperature";

**parameter** ConvectionCoefficient h=0.7 "Convective cooling coefficient";

**parameter** Area A=1.0 "Surface area";

**parameter** Mass m=0.1 "Mass of thermal capacitance";

**parameter** SpecificHeat c\_p=1.2 "Specific heat";

*// Variables*

Temperature T "Temperature";

**initial equation**

T = T0 "Specify initial value for T";

**equation**

m\*c\_p\*der(T) = h\*A\*(T\_inf-T) "Newton's law of cooling";

**end** **NewtonCoolingWithTypes**;

You can read the definition type Temperature=Real(unit="K", min=0); as “Let us define a new type, Temperature, that is a specialization of the built-in type Real with physical units of Kelvin (K) and a minimum possible value of 0”.

From this example, we can see that once we define a physical type like Temperature, we can use it to declare multiple variables (*e.g.*, T, T\_inf and T0) without having to specify the unit or min attribute for each variable. Also, we get to use the familiar name Temperature instead of the SI unit, K. You might be wondering what other attributes are available when creating derived types. For further discussion, see the section on [*Built-In Types*](http://book.xogeny.com/behavior/equations/variables/#builtin-types).

At this point, you might find the idea of defining Temperature, ConvectionCoefficient, SpecificHeat and Mass in every model extremely tedious. It would be, if it were truly necessary. But don’t worry, there is an easy solution to this as you will see in a later section where we discuss [*Importing Physical Types*](http://book.xogeny.com/components/packages/nimport/#importing-physical-types).