

---

## AIMMS Language Reference - Units of Measurement

This file contains only one chapter of the book. For a free download of the complete book in pdf format, please visit [www.aimms.com](http://www.aimms.com) or order your hard-copy at [www.lulu.com/aimms](http://www.lulu.com/aimms).

Copyright © 1993–2011 by Paragon Decision Technology B.V. All rights reserved.

Paragon Decision Technology B.V.	Paragon Decision Technology Inc.	Paragon Decision Technology Pte.
Schipholweg 1	500 108th Avenue NE	Ltd.
2034 LS Haarlem	Ste. # 1085	80 Raffles Place
The Netherlands	Bellevue, WA 98004	UOB Plaza 1, Level 36-01
Tel.: +31 23 5511512	USA	Singapore 048624
Fax: +31 23 5511517	Tel.: +1 425 458 4024	Tel.: +65 9640 4182
	Fax: +1 425 458 4025	

Email: [info@aimms.com](mailto:info@aimms.com)  
WWW: [www.aimms.com](http://www.aimms.com)

AIMMS is a registered trademark of Paragon Decision Technology B.V. IBM ILOG CPLEX and sc CPLEX is a registered trademark of IBM Corporation. GUROBI is a registered trademark of Gurobi Optimization, Inc. KNITRO is a registered trademark of Ziena Optimization, Inc. XPRESS-MP is a registered trademark of FICO Fair Isaac Corporation. MOSEK is a registered trademark of Mosek ApS. WINDOWS and EXCEL are registered trademarks of Microsoft Corporation.  $\text{T}_{\text{E}}\text{X}$ ,  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ , and  $\mathcal{A}\mathcal{M}\mathcal{S}\text{-}\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  are trademarks of the American Mathematical Society. LUCIDA is a registered trademark of Bigelow & Holmes Inc. ACROBAT is a registered trademark of Adobe Systems Inc. Other brands and their products are trademarks of their respective holders.

Information in this document is subject to change without notice and does not represent a commitment on the part of Paragon Decision Technology B.V. The software described in this document is furnished under a license agreement and may only be used and copied in accordance with the terms of the agreement. The documentation may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Paragon Decision Technology B.V.

**Paragon Decision Technology B.V. makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Paragon Decision Technology B.V., its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claims for lost profits, fees or expenses of any nature or kind.**

**In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors shall not be responsible under any circumstances for providing information or corrections to errors and omissions discovered at any time in this book or the software it describes, whether or not they are aware of the errors or omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors do not recommend the use of the software described in this book for applications in which errors or omissions could threaten life, injury or significant loss.**

This documentation was typeset by Paragon Decision Technology B.V. using  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  and the LUCIDA font family.

**Part VII**

---

**Advanced Language  
Components**

## Chapter 30

# Units of Measurement

This chapter describes how to incorporate dimensional analysis into an AIMMS application. As will be explained, you can define quantities and their corresponding units, and associate these units with identifiers in your model. AIMMS automatically checks for unit consistency in all the constraints and assignment statements. In addition, AIMMS allows you to specify unit conventions. With this facility it is possible for end-users around the world to select their preferred convention, and view the model data in the units associated with that convention.

*This chapter*

---

### 30.1 Introduction

Measurement plays a central role in observations of the real world. Most observed quantities are measured in some unit (e.g. dollar, hour, meter, etc.), and the magnitude of the unit influences the mental picture that you may have of an object (e.g. ounce, kilogram, ton, etc.). When you combine such objects in a numerical relationship, the corresponding units must be *commensurable*. Without such consistency, the mathematical relationships become meaningless.

*Units are common*

There are several good reasons to track units throughout a model. The explicit mentioning of units can enhance the readability of a model, which is especially helpful when others read and/or maintain your model. Units provide the AIMMS compiler with additional checking power to find errors in model formulations. Finally, through the use of units you can let AIMMS perform the job of unit conversion and scaling.

*Why units in models*

The model editor in AIMMS will give you access to a large number of quantities and units, and in particular to those of the International System of Units (referred to as SI from the French “Systeme Internationale”). The SI system is an improved metric system adopted by the Eleventh General Conference of Weights and Measures in 1960. The entire SI system of measurement is constructed from the atomic base units associated with the following nine basic quantities.

*Standard units*

Quantity	Atomic Base Unit	Text
length	m	meter
mass	kg	kilogram
time	s	second
temperature	K	kelvin
amount of mass	mol	mole
electric current	A	ampere
luminous intensity	cd	candela
angle	rad	radian
solid angle	sr	steradian

Table 30.1: Basic SI quantities and their base units

All quantities which are not one of the nine basic SI quantities are called *derived* quantities. Each such quantity has a derived base unit which can be expressed in terms of the atomic base units of the basic SI quantities. Optionally, a compound unit symbol can be associated with such a derived base unit, like the symbol N for the unit  $\text{kg}\cdot\text{m}/\text{s}^2$ . The following table illustrates some of the more well-known derived quantities and their corresponding derived base units. Note that five of them have an associated compound unit symbol. Many other derived quantities are available in AIMMS.

*Derived quantities and units*

Quantity	Derived Base Unit	Text
area	$\text{m}^2$	square meter
volume	$\text{m}^3$	cubic meter
force	$\text{N} = \text{kg}\cdot\text{m}/\text{s}^2$	newton
pressure	$\text{Pa} = \text{kg}/\text{m}\cdot\text{s}^2$	pascal
energy	$\text{J} = \text{kg}\cdot\text{m}^2/\text{s}^2$	joule
power	$\text{W} = \text{kg}\cdot\text{m}^2/\text{s}^3$	watt
charge	$\text{C} = \text{A}\cdot\text{s}$	coulomb
density	$\text{kg}/\text{m}^3$	kilogram per cubic meter
velocity	$\text{m}/\text{s}$	meter per second
angular velocity	$\text{rad}/\text{s}$	radian per second

Table 30.2: Selected derived SI quantities and their base units

Aside from the base unit that must be associated with every quantity, it is also possible to specify a number of *related* units. Related units are those units that can be expressed in terms of their base unit by means of a linear relationship. A typical example is the unit km which is related to the base unit m by means of the linear relationship  $x \text{ km} = 1000 \cdot x \text{ m}$ . Similarly, the unit degC (degree Celsius) is related to the base unit K through the formula  $x \text{ degC} = (x + 273.15) \text{ K}$ .

*Related units*

Frequently, related units are a multiple of their own base unit, which is reflected through a prefix notation that indicates the level of scaling. Table 30.3 shows the standard SI prefix symbols and their corresponding scaling factor. Familiar examples are kton, MHz, kJ, etc. Note that any prefix can be applied to any base unit except the kilogram. The kilogram takes prefixes as if the base unit were the gram.

*Standard unit  
prefix notation*

Factor	Name	Symbol	Factor	Name	Symbol
$10^1$	deca	da	$10^{-1}$	deci	d
$10^2$	hecto	h	$10^{-2}$	centi	c
$10^3$	kilo	k	$10^{-3}$	milli	m
$10^6$	mega	M	$10^{-6}$	micro	mu
$10^9$	giga	G	$10^{-9}$	nano	n
$10^{12}$	tera	T	$10^{-12}$	pico	p
$10^{15}$	peta	P	$10^{-15}$	femto	f
$10^{18}$	exa	E	$10^{-18}$	atto	a
$10^{21}$	zetta	Z	$10^{-21}$	zepto	z
$10^{24}$	yotta	Y	$10^{-24}$	yocto	y

Table 30.3: Prefixes of the International System

To give you maximum freedom to choose your own quantities, units and naming conventions, AIMMS is not exclusively committed to any particular standard. However, you are encouraged to use the standard SI units and prefix symbols to make your model as readable and maintainable as possible.

*Flexible  
specification*

Thus far you have encountered *basic* quantities (Table 30.1) and *derived* quantities (Table 30.2). Each quantity has a *base* unit. The base unit of a basic quantity is defined through a unit symbol, referred to as an *atomic* unit. All other base units are *derived* base units. Such units are defined through an expression in terms of other base units, which can eventually be translated into an expression of atomic base units. You have the option to associate a unit symbol with any derived base unit, which is referred to as a *compound* unit symbol. Whenever you have associated a unit symbol with the base unit of either a basic or derived quantity, you are also allowed to specify one or more *related* unit symbols by specifying the corresponding linear relationship.

*Summary of  
terminology*

---

## 30.2 The QUANTITY declaration

In AIMMS, all units are uniquely coupled to declared quantities. For each declared QUANTITY you must specify an identifier together with one or more of its attributes listed in Table 30.4.

*Declaration*

Attribute	Value-type	Mandatory
BASE UNIT	[ <i>unit-symbol</i> ] [=] [ <i>unit-expression</i> ]	yes
TEXT	<i>string</i>	
CONVERSION	<i>unit-conversion-list</i>	
COMMENT	<i>comment string</i>	

Table 30.4: Quantity attributes

You must always specify a base unit for each quantity that you declare. Its value is either

*The BASE UNIT attribute*

- an *atomic* unit symbol,
- a *unit expression*, or
- a *compound* unit symbol with unit expression.

A unit symbol can be any sequence of the characters a-z, the digits 0-9, and the symbols `_`, `@`, `&`, `%`, `|`, as well as a currency symbol not starting with a digit, or one of the special unit symbols `1` and `-`. The latter two special unit symbols allow you, for instance, to declare model identifiers without unit, or to express unitless numerical data in terms of percentages.

The Ascii version of AIMMS supports the following currency symbols: \$, €, ¢, £, ¤, ¥. The Unicode version of AIMMS supports the currency symbols as defined by the Unicode committee which include the above currency symbols (see also <http://unicode.org/charts/PDF/U20A0.pdf>)

*Currency symbols*

AIMMS stores unit symbols in namespaces separate but parallel to the identifier namespaces. Hence, you are free to choose unit symbols equal to the names of global identifiers within your model. Namespaces in AIMMS are discussed in full detail in Section 33.4.

*Separate namespace*

AIMMS 3.8 and older use only a singleton unit namespace which was a potential cause of nameclashes when units with the same name are declared from quantities or unit parameters declared in different namespaces. In order to obtain the old behaviour one can make sure that all units are declared within the global namespace or set the option `singleton unit namespace` to `on`. This option can be found in the backward compatibility category.

*Backward compatibility*

The following example illustrates the three types of base units.

*Example*

```

QUANTITY:
  identifier : Length
  base unit : m ;
QUANTITY:
  ! atomic unit

```

```

    identifier : Time
    base unit : s ;                ! atomic unit
QUANTITY:
    identifier : Velocity
    base unit : m/s ;            ! unit expression
QUANTITY:
    identifier : Frequency
    base unit : Hz = 1/s ;       ! compound unit symbol with unit expression

```

The atomic unit symbols `m` and `s` are the base units for the quantities Length and Time. The unit expression `m/s` is the base unit for the quantity Velocity. The compound unit symbol `Hz`, defined by the unit expression `1/s`, is the base unit of the quantity Frequency.

The previous example strictly adheres to the SI standards, and, for example, defines the base unit of the derived quantity Frequency in terms of the base unit of Time. In general, this is not necessary. If Time is not used anywhere else in your model, you can just provide the base unit `Hz` for Frequency without providing its translation in SI base units. Frequency then becomes a basic quantity, and `Hz` becomes an atomic base unit.

*Derived can be used as basic*

The unit expressions that you can enter in the BASE UNIT attribute can only consist of

*Unit expressions*

- unit symbols (base and/or related units),
- constant factors,
- the two operators “\*” and “/”,
- parentheses, and
- the power operator “^” with integer exponent.

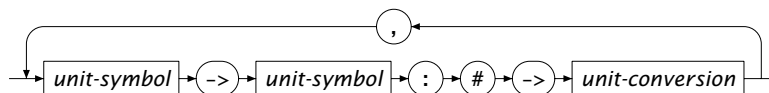
The common precedence order of the operators “\*”, “/” and “^” is as described in Section 6.3. Unit expressions are discussed in full detail in Section 30.6.

With the CONVERSION attribute you can declare and define one or more related unit symbols by specifying the (linear) transformation to the associated base unit. The conversion syntax is as follows.

*The CONVERSION attribute*

*unit-conversion-list :*

*Syntax*



A unit conversion must be defined using a linear expression of the form  $(\# \cdot a + b)$  where `#` is a special token, and the operator `·` stands for either multiplication or division. The coefficients `a` and `b` can be either numerical constants or references to scalar parameters. An example in which the use of scalar

*Unit conversion explained*

parameters is particularly convenient is the conversion between currencies parameterized by a varying exchange rate.

*Example*

```

QUANTITY:
  identifier : Length
  base unit  : m
  conversions : km  -> m   : # -> # * 1000 ,
                mile -> m   : # -> # * 1609 ;

QUANTITY:
  identifier : Temperature
  base unit  : degC
  conversions : degC -> degF : # -> # * 1.8 + 32 ;

QUANTITY:
  identifier : Energy
  base unit  : J = kg * m^2 / s^2
  conversions : kJ  -> J   : # -> # * 1000 ,
                MJ  -> J   : # -> # * 1.0e6 ,
                kWh -> J   : # -> # * 3.6e6 ;

QUANTITY:
  identifier : Currency
  base unit  : US$
  conversion : DM  -> US$ : # -> # * ExchangeRate('DM') ,
                DFl -> US$ : # -> # * ExchangeRate('DFl') ;

QUANTITY
  identifier : Unitless
  base unit  : 1
  conversions : % -> 1    : # -> # / 100 ;

```

---

### 30.3 Associating units with model identifiers

To associate units with scalar or multi-dimensional identifiers in your model, you can specify a unit definition for such identifiers through the UNIT attribute. The UNIT attribute is only supported for the following identifier types:

*The UNIT attribute*

- parameters,
- variables,
- constraints,
- arcs,
- nodes,
- function and procedure arguments, and
- internal and external functions.

Within the AIMMS **Model Explorer**, the UNIT attribute is only visible in the attribute forms of the identifier types listed above if your model already contains the declarations of one or more quantities. If you only want to use the UNIT attribute to specify a scale factor for an identifier (see below), you can make the UNIT attribute visible in all attribute forms by adding a *unitless* quantity to your model (i.e. a quantity with base unit 1).

*Visibility of the UNIT attribute*

In its simplest form, the unit definition of a parameter, variable or constraint is just a reference to a base or compound unit symbol. In general, it can be a unit expression based on the same syntax as described previously for specifying a derived unit expression in a QUANTITY declaration. The complete syntax of unit expressions is discussed in Section 30.6.

*UNIT attribute value*

The declaration

*Example*

```
VARIABLE:
  identifier : VelocityOfItem
  index domain : i
  unit      : km/h ;
```

introduces a variable VelocityOfItem(i) with a corresponding unit km/h. This declaration could also have been written as

```
VARIABLE:
  identifier : VelocityOfItem
  index domain : i
  unit      : 1000*m/h ;
```

which contains an explicit scale factor of 1000, instead of using the derived unit symbol km.

When you do not use unit symbols, you can still use the UNIT attribute to indicate the appropriate scale factor to be used for an identifier. These scale factors, whether or not in the presence of unit symbols, will be used by AIMMS to scale the corresponding data during various computations, as explained in Section 30.5.

*Units also for scaling*

By specifying units for some or all the identifiers in your model, AIMMS will perform the following unit-related tasks for you:

*Use of units*

- automatic checking of the statements in your model for unit consistency (see Section 30.4),
- automatic scaling of identifiers in assignments, DISPLAY and READ/WRITE statements (see Section 30.5), and
- automatic conversion of arguments (and result value) of external procedures and functions (see Section 30.5), and
- automatic scaling of the variables and constraints in a mathematical program (see Section 30.5.1).

For all identifier types for which you can specify a UNIT attribute, there is also an associated .Unit suffix. The value of the .Unit suffix is a unit expression that equals the unit specified within the UNIT attribute of the identifier at hand.

*The .Unit suffix*

The `.Unit` suffix is most commonly used in the following situations:

- when generating reports by means of the `PUT` and `DISPLAY` statements (see Sections 29.2 and 29.3, respectively),
- when displaying units in strings generated by the `%u` conversion specifier of the `FormatString` function (see Section 5.3.2), and
- when performing sensitivity analysis of mathematical programs in the presence of variables and constraints which have a non-empty `UNIT` attribute (see Section 30.5.1).

*Use of the `.Unit` suffix*

If you want to reference the `.Unit` suffix of a multidimensional identifier, it is not always necessary to use the corresponding indices of the identifier in its `.Unit` suffix reference. The use of indices is only necessary if the `UNIT` attribute actively depends on the indices, for instance, because it

*Indices not always required*

- contains a multidimensional scale factor, or
- refers to a multidimensional unit parameter (see also Section 30.9).

In all other cases, a reference to just the identifier name is sufficient.

Consider the declaration of the variable `VelocityOfItem(i)` above. Its `UNIT` attribute is the constant unit `km/h`, whence it can be obtained through the (scalar) reference

*Example*

```
VelocityOfItem.Unit
```

When the `UNIT` attribute of an identifier contains references to *unit-valued* parameters (see Section 30.9), such references will be evaluated, within the context of the `.Unit` suffix, to their corresponding unit expressions. Thus, the `.Unit` suffix will always result in a unit expression containing only unit symbols declared in one or more `QUANTITY` declarations.

*Unit-valued parameters are permitted*

---

## 30.4 Unit analysis

By associating a unit with every relevant identifier in your model, you enable AIMMS to automatically verify whether all terms in the assignments and constraints of your model are unit consistent. When AIMMS detects unit inconsistencies, this may help you to solve conceptual problems in your model, which could otherwise have remained undetected for a long time.

*Unit consistency*

With every derived unit or compound unit symbol, it is possible to associate a unique unit expression consisting of a constant scale factor and atomic units only. All assignments and definitions in AIMMS are interpreted as formulas expressed in terms of these atomic unit expressions, and unit consistency checking is based on this interpretation. While ignoring the constant scale factors, AIMMS will verify that the atomic unit expression for every term in either an assignment statement or a constraint is identical. If the resulting unit check identifies an inconsistency, an error or warning will be generated.

*... always in atomic units*

Consider the identifiers a, b, and c having units [m], [km], and [10\*m] respectively, all with [m] as their corresponding associated atomic unit expression, and scale factors 1, 1000 and 10, respectively. Then the assignment

*Example*

```
c := a + b ;
```

is *unit consistent*, because all terms share the same atomic unit expression [m].

If an expression on the right-hand side of an assignment consists of a constant scalar term or a data expression (preceded by the keyword DATA), AIMMS will assume by default that such expressions have the same unit as the identifier on the left-hand side. If the intended unit of the right-hand side is different than the declared unit of the identifier on the left, you should explicitly specify the appropriate unit for this term, by locally overriding the unit as explained in Section 30.7.

*Constant expressions*

On the other hand, if a non-constant expression contains a constant term, then AIMMS will make no assumption about the intended unit of the constant term. In fact, it is considered unitless. If a unit inconsistency occurs for that reason, you should explicitly add a unit to the constant term to resolve the inconsistency, as explained in Section 30.7.

*Constant terms in expressions*

Given parameters a ([m]) and b ([km]), as well as a 1-dimensional parameter d(i) with associated unit [m], the following assignments illustrate the interpretation of constant numbers by AIMMS.

*Example*

```
a := 10;                ! OK: constant number 10 interpreted as [m]
a := 10 [km];          ! OK: constant number 10 interpreted as [km]
d(i) := DATA { 1: 10, 2: 20 }; ! OK: all data interpreted as [m]
a := 10*b;             ! OK: constant number 10 considered unitless
a := b + 10;           ! ERROR: unit inconsistency, constant term 10 unitless
a := b + 10 [km];     ! OK: unit inconsistency resolved
```

By default, the global AIMMS option to perform automatic unit analysis is on and inconsistencies are detected. AIMMS will produce either warning messages or error messages (the former is the default). You can find the full details on all unit-related options in the help file that comes with your AIMMS system.

*Automatic unit checking on or off*

The assignment `c := a + b` of the first example in this section is unit consistent, but it does not appear to be *scale consistent* since the units of `a`, `b` and `c` have different scales. In AIMMS, however, a unit consistent assignment is automatically scale consistent, because AIMMS translates and stores all data in terms of the underlying atomic unit expression. In the example, this implies that the use of the values of `a`, `b`, and `c` as well as the assignment are in the atomic unit [m]. Consequently, AIMMS can now directly execute the assignment, and the scale consistency is automatically ensured. Of course, any *display* of values of `a`, `b` and `c` will be again in terms of the units associated with these identifiers.

*Automatic scale consistency*

This example illustrates a number of identifiers with compound unit definitions. It is based on the SI units for weight, velocity and energy, and uses the derived units ton, km, h and MJ.

*Advanced example...*

```
VARIABLE:
  identifier : WeightOfItem
  index domain : i
  unit      : ton ;
VARIABLE:
  identifier : VelocityOfItem
  index domain : i
  unit      : Velocity: km/h ;
VARIABLE:
  identifier : KineticEnergyOfItem
  index domain : i
  unit      : MJ
  definition : 1/2 * WeightofItem(i) * VelocityOfItem(i)^2 ;
```

Any display of these variables will be in terms of ton, km/h and MJ, respectively, but internally AIMMS uses the units kg, m/s and  $\text{kg}\cdot\text{m}^2/\text{s}^2$  for storage. The latter represent the corresponding unique atomic unit expressions associated with weight, velocity and energy.

As a consequence of specifying units, there will be an automatic consistency check on the defined variable `KineticEnergyOfItem(i)`. AIMMS interprets the definition of `KineticEnergyOfItem(i)` as a formula expressed in terms of the atomic units. The relevant unit components are:

*... is unit consistent*

- [ton ] =  $10^3$  \* [kg ],
- [km/h] = (1/3.6) \* [m/s ], and
- [MJ ] =  $10^6$  \* [ $\text{kg}\cdot\text{m}^2/\text{s}^2$ ].

The definition of `KineticEnergyOfItem(i)` as expressed in terms of atomic units is  $\text{kg} \cdot (\text{m}/\text{s})^2$ , while its own unit in terms of atomic units is  $\text{kg} \cdot \text{m}^2/\text{s}^2$ . These two unit expressions are consistent.

If the unit conversion between a derived unit and its corresponding atomic unit not only consists of a scale factor, but also contains a constant term, such a derived unit is referred to as a *non-absolute* unit. If an arithmetic expression in your model refers to identifiers or constants expressed in a non-absolute unit, you should pay special attention to make sure that the result of the computation is what you intended. The following example makes the point.

*Beware of  
non-absolute  
units*

Consider the following quantity declaration.

*Example*

```
QUANTITY:
  identifier : Temperature
  base unit  : K
  conversions : degC -> K : # -> # + 273.15 ;
```

Given this declaration, what is the result of the assignment

```
x := 1 [degC] + 2 [degC];
```

where `x` is a scalar parameter with unit `degC`? Following the rules explained above—AIMMS stores all data and performs all computations in terms of atomic units— AIMMS performs the following computation internally

```
x := 274.15 [K] + 275.15 [K];
```

resulting in an assignment to `x` of  $549.3 \text{ [K]} = 276.15 \text{ [degC]}$ , which is probably not the intended answer. The key observation is that in an addition only one of the operands should be expressed in a non-absolute unit. Similarly, in a multiplication or division probably none of the operands should be expressed in a non-absolute unit. The mistake in the above assignment is that the second argument in fact should be a temperature difference (e.g. between  $3 \text{ [degC]}$  and  $1 \text{ [degC]}$ ), which precisely yields an expression in terms of the corresponding absolute unit `K`:

```
x := 1 [degC] + (3 [degC] - 1 [degC]);    ! equals 274.15 [K] + 2 [K] = 3 [degC]
```

Using temperature differences is more common in assignments to identifiers like `LengthIncreasePerDegC` (expressed in `[m/degC]`), which probably takes the form of a *difference quotient*, as illustrated below.

```
LengthIncreasePerDegC := (Length1 - Length0) / (Temperature1 - Temperature0);
```

When you use an intrinsic AIMMS function (see Section 6.1.4) inside an expression in your model, the unit associated with the corresponding function call will in general depend on its arguments. The unit relationship between the arguments and the result of the function falls into one of the following function categories.

*Units and intrinsic functions*

- *Unitless* functions, for which both the arguments and the result are dimensionless. Examples are: exp, log, log10, errorf, atan, cos, sin, tan, degrees, radians, atanh, cosh, sinh, tanh, and the exponential operator with a non-constant exponent.
- *Transparent* functions that do not alter units. Examples are: abs, max, min, mod, ceil, floor, precision, round, and trunc.
- *Conversion* functions that convert units in a predictable way. Examples are: sqr, sqrt, and the exponential operator with a constant integer exponent.

In some exceptional cases, one or more terms in an expression may not be unit consistent with the other terms in the expression. To restore unit consistency, AIMMS allows you to explicitly specify a unit for the inconsistent term(s) as an emergency measure. The syntax for such unit overrides is explained in Section 30.7. You should make sure, however, that these explicit unit overrides do not affect the scale consistency of the expression (see Section 30.7).

*Explicit units in expressions*

---

### 30.4.1 Unit analysis of procedures and functions

Once you have associated units of measurement with the global identifiers in your model, you will also need to associate units of measurement with the arguments, local identifiers and result values of procedures and functions. When you do so, you enable AIMMS to perform the common unit analysis on the statements in the bodies of all internal procedures and functions. For external procedures and functions, AIMMS cannot perform a unit analysis on the function and procedure bodies, but will use the assigned units for scaling purposes as explained in Section 30.5.

*Unit analysis of procedures and functions*

In general, one can distinguish two types of procedures and functions, namely

*Two procedure types*

- procedures and functions of a very specific nature, whose arguments and result values have associated units of measurement that are constant and known a priori, and
- procedures and functions of a very general nature, whose arguments and result values can have any associated unit of measurement.

An example of the latter type is a function with a single one-dimensional argument to compute the average of all values contained in its argument. For such a function, the specific units associated with the argument and the result values are not known a priori, but it is known that they must be equal.

To let you declare procedure and functions of the second type, AIMMS allows you to express the units of measurement of its arguments and the result values in terms of unit parameters (see also Section 30.9) declared locally within the procedure or function. At runtime, AIMMS will dynamically determine the value of the unit parameter, based on the actual arguments passed to the procedure or function. In addition, AIMMS will verify that the unit of a function value is commensurate with the remainder of the statement or expression from which it was called.

*Express units  
in local unit  
parameters*

The function `MyAverage` in this example computes the average of a general one-dimensional identifier. It combines AIMMS' ability to define arguments over local sets (see Section 10.1), with a unit expressed in term of a local unit parameter. Its declaration is given by

*Example*

```
FUNCTION
  identifier: MyAverage
  arguments : (Ident)
  unit      : LocalUnit
  body      :
    MyAverage := sum(i, Ident(i)) / Card(LocalSet);
```

The single argument `Ident(i)` of the function `MyAverage` is defined by

```
PARAMETER:
  identifier : Ident
  index domain : i
  unit      : LocalUnit ;
SET:
  identifier : LocalSet
  index     : i ;
UNIT PARAMETER:
  identifier : LocalUnit ;
```

Note that `Ident(i)` is defined over a local set `LocalSet` and that its unit is expressed in terms of a local unit parameter `LocalUnit`, both of which are determined at runtime. Because the unit of the function `MyAverage` itself is also equal to `LocalUnit`, the assignment in the body of `MyAverage` is unit consistent.

---

## 30.5 Unit-based scaling

With each identifier for which you have specified a UNIT attribute, AIMMS associates two values:

*Scaled versus  
unscaled values*

- the *scaled* value (i.e. expressed in terms of the unit specified), and
- the *unscaled* value (i.e. expressed in terms of the associated atomic unit expression).

The transformation between scaled and nonscaled values is completely determined by the product of explicit and implicit scale factors associated with the various quantity and unit definitions.

As mentioned in Section 30.4, AIMMS uses internally unscaled values for all storage and arithmetic computations. This guarantees automatic scale consistency. However, for external use, scaled values are more natural when exchanging data with components outside the AIMMS execution system. Specifically, AIMMS uses scaled values when

*When used*

- displaying the data of an identifier in the (end-)user interface,
- exchanging data for a particular identifier with files and databases using the READ and WRITE statements,
- passing arguments to external procedures and functions,
- storing the result value(s) of an external function, and
- communicating the variables and constraints of a mathematical program to a solver.

When displaying data in either the graphical user interface or in PUT and DISPLAY statements, AIMMS will transfer data using the scaled unit specified in the definition of the identifier. For example, if you have specified kton as the unit attribute of an identifier while the underlying atomic unit is kg, AIMMS will still display the identifier values in kton.

*Units in displays*

Similarly, when reading data from or writing data to scalar numerical constants, lists, tables, composite tables (either graphical or in data files), or tables (in databases) using the READ and WRITE statements, AIMMS assumes that this data is provided in the (scaled) units that you have specified in the identifier declarations in your model, and will transform all data to the corresponding unscaled values for internal storage.

*... and data entry*

You can override the default scaling based on the content of the UNIT attribute either locally within the graphical end-user interface or model source, or globally using CONVENTIONS. Local and global overrides are discussed in complete detail in Sections 30.7 and 30.8.

*Override default scaling*

---

### 30.5.1 Unit-based scaling of mathematical programs

During communications with a solver, AIMMS will scale all variables and constraints (including variable definitions) in accordance with the scale factor associated with the UNIT attribute in their declaration. This choice is based on the assumption that the specified units reflect the expected order of magnitude of the numbers associated with the variables, parameters and constraints, and that these numbers will neither be very large nor very small. As a result, the values of all rows and columns in the generated mathematical program are expected to be of the same, reasonable, order of magnitude. Especially nonlinear solvers may greatly benefit from this choice.

*Automatic scaling for solvers*

In the main example of Section 30.4, the scale factors are  $10^3$  for the identifier `WeightOfItem(i)`,  $1/3.6$  for `VelocityOfItem(i)`, and  $10^6$  for `KineticEnergyOfItem(i)`. The entire constraint associated with the defined variable is then scaled according to the scale factor of the unit of the definition variable `KineticEnergyOfItem(i)`, MJ. This corresponds with dividing the left- and right-hand side of the constraint by  $10^6$ . Thus, the resulting expression communicated to the solver by AIMMS will be:

*Main example revisited*

```
KineticEnergyOfItemColumn(i) =
  1/2 * (1/10^3) * WeightOfItemColumn(i)
  * ((1/3.6) * VelocityOfItemColumn(i))^2 ;
```

Notice that each variable shown in this expression has a suffix “Column” to indicate that it corresponds to a column in the matrix underlying the mathematical program.

Some care is needed when you have requested sensitivity information associated with a mathematical program, such as the reduced costs of variables and shadow prices of constraints. The basic rules with respect to retrieving sensitivity information are as follows:

*Units of reduced cost and shadow price*

- All sensitivity suffices in AIMMS, such as the `.ReducedCost` and `.ShadowPrice` suffix, are unitless.
- All sensitivity suffices hold the exact numerical value as computed by the solver, i.e. expressed with respect to the scaled values that are communicated to the solver by AIMMS.

The reason for not associating units with the sensitivity suffices is that a single variable or constraint may be used in multiple mathematical programs, each with its own objective. As each objective may have a different associated unit, and the reduced costs and shadow prices express properties of a variable or constraint with respect to the objective, it is inherently impossible to associate a single unit with the `.ReducedCost` and `.ShadowPrice` suffices.

*Motivating the choice of unitless*

You may encounter scaling problems when you want to perform direct computations with the sensitivity suffices of variables and constraints. Using the `.Unit` suffix and AIMMS’ capabilities to override units of subexpressions (see Sections 30.6 and 30.7), however, it is easy to formulate expressions that

*Unit- and scale consistent sensitivity data*

- result in the correct unscaled numerical values that can be used directly in AIMMS computations, and
- have an associated unit that is consistent with their interpretation.

Assuming that `ExampleVariable` and `ExampleConstraint` are part of a mathematical program, with `ObjectiveVariable` as its objective function, one can obtain the correct values by locally overriding the units of the `.ReducedCost` and `.ShadowPrice` suffices through the expressions:

*Example with unit overrides*

```
(ExampleVariable.ReducedCost ) [ObjectiveVariable.Unit / ExampleVariable.Unit ]
(ExampleConstraint.ShadowPrice) [ObjectiveVariable.Unit / ExampleConstraint.Unit]
```

Alternatively, you can use the function `EvaluateUnit` (see Section 30.6.2) to obtain the same result

*Example with unit functions*

```
ExampleVariable.ReducedCost *
    EvaluateUnit( ObjectiveVariable.Unit / ExampleVariable.Unit )
ExampleConstraint.ShadowPrice *
    EvaluateUnit( ObjectiveVariable.Unit / ExampleConstraint.Unit )
```

If you need to perform multiple computations with these expressions, or want to display them in the graphical end-user interface, you are advised to assign these expressions to additional parameters in your model with the appropriate associated units.

*Introducing new parameters*

When you have used a CONVENTION to override the default scaling during the SOLVE statement, the expressions above should be augmented by applying the functions `ConvertUnit` and `EvaluateUnit` (see Section 30.6.1):

*Example with convention*

```
ExampleVariable.ReducedCost *
    EvaluateUnit( ConvertUnit(ObjectiveVariable.Unit, ConventionUsed) /
                  ConvertUnit(ExampleVariable.Unit, ConventionUsed) )
ExampleConstraint.ShadowPrice *
    EvaluateUnit( ConvertUnit(ObjectiveVariable.Unit, ConventionUsed) /
                  ConvertUnit(ExampleConstraint.Unit, ConventionUsed) )
```

This will result in a scaling factor that is consistent with the variable and constraint scaling convention passed to the solver. You cannot obtain the same result by locally overriding the units of the `.ReducedCost` and `.ShadowPrice` suffices, as unit local overrides only accept simple unit expressions (see Section 30.6).

If your model contains multiple computations concerning the `.ReducedCost` and `.ShadowPrice` suffices, each with identical scale factors, you may consider assigning the unit expressions required for scaling these suffices to unit parameters (see Section 30.9). You can then directly use such unit parameters in a local unit override, rather than having to repeat possibly complex unit expressions time and again. For instance, if `ScaledUnit` is a unit parameter defined by

*Use of unit parameters*

```
ScaledUnit := ConvertUnit(ObjectiveVariable.Unit, ConventionUsed) /
              ConvertUnit(ExampleVariable.Unit, ConventionUsed) ;
```

then the correctly scaled expression for the reduced cost of ExampleVariable can be simplified to

```
(ExampleVariable.ReducedCost) [SCaledUnit]
```

You can use a local override, because a reference to a scalar unit parameter again forms a valid simple unit expression (see Section 30.6).

## 30.6 Unit expressions

Unit expressions can be used at various places in an AIMMS model, such as:

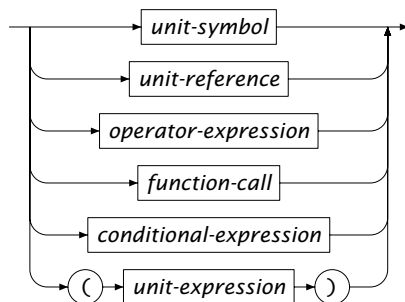
*Unit expressions*

- the BASE UNIT attribute of a QUANTITY declaration (defined in Section 30.2),
- in a local unit override of a numerical (sub-)expression (discussed in Section 30.7)
- in a convention list of the PER UNIT, PER QUANTITY or PER IDENTIFIER attributes of a CONVENTION (see also Section 30.8), or
- on the right hand side of an assignment to a unit parameter (see Section 30.9).

The syntax of a unit expression is straightforward, and given below.

*unit-expression* :

*Syntax*



The simplest form of unit expression is just a unit symbol, as defined in either the BASE UNIT or the CONVERSION attribute of a QUANTITY declaration. A reference to either a (scalar or indexed) unit parameter (see Section 30.9) or to the .Unit suffix of any identifier with an associated unit (see Section 30.3), is a second form of unit expression.

*Unit symbols and references*

More complex unit expressions can be obtained by applying the binary unit operators \*, / and ^, with the usual left-to-right evaluation order. The following rules apply:

*Unit operators and functions*

- the operand on the right of the `*` operator must be a unit expression, while the operand on the left can either be a unit expression or a numerical expression (expressing a numeric scale factor),
- both operands of the `/` operator must be unit expressions, and
- the operand on the left of the `^` operator must be a unit expression, while the exponent operand must be an integer numerical expression.

In addition, AIMMS supports a number of unit functions, which can create new unit values or construct associated unit values from a given unit expression (see Section 30.6.1).

However, AIMMS requires that any unit expressions uniquely falls into one of the three categories

*Three types of unit expressions*

- unit constant,
- simple unit expression, or
- computed unit expression.

*Unit constants* are unit expressions which consist solely of unit symbols, scalar constants and the three unit operators `*`, `/` and `^`. Unit constants can be used in

*Unit constants*

- the `BASE UNIT` attribute of a `QUANTITY`,
- the lists associated with a `CONVENTION`, and
- the unit-valued function `Unit`.

In addition, unit constants can be

- displayed and entered via the AIMMS graphical user interface,
- assigned to unit parameters through data statements (see Chapter 26), and
- exchanged with external data sources via the `READ` and `WRITE` statements (see Chapter 24).

*Simple unit expressions* are an extension of unit constants. They are unit expressions which consist solely of unit symbols, unit references without indexing, scalar constants and the three unit operators `*`, `/` and `^`. Simple unit expressions can be used in

*Simple unit expressions*

- local unit overrides, and
- assignments to unit parameters.

Computed unit expression can use the full range of unit expressions, with the exception of unit constants. If you want to refer to unit constants within the context of a computed unit expression, you must embed it within a call to the function `Unit`, discussed in the next section. Computed unit expressions can be used

*Computed unit expressions*

- in assignments to unit parameters, and
- as an argument of the functions `ConvertUnit`, `AtomicUnit` and `EvaluateUnit` (see Sections 30.6.1 and 30.6.2).

---

### 30.6.1 Unit-valued functions

AIMMS supports the following unit-valued functions:

*Unit-valued  
functions*

- `Unit(unit-constant)`
- `StringToUnit(unit-string)`
- `AtomicUnit(unit-expr)`
- `ConvertUnit(unit-expr, convention)`

The function `Unit` simply returns its argument, which must be a unit constant. The function `Unit` is available to allow the usage of unit constants within computed unit expressions (as discussed in the previous section).

*The function  
Unit*

The function `StringToUnit` converts a string, which represents a unit expression, to the corresponding unit value. You can use this function, for instance, after reading external string data that needs to be converted to real unit values for further use in your model.

*The function  
StringToUnit*

With the function `AtomicUnit` you can retrieve the atomic unit expression corresponding to the unit expression passed as the argument to the function. Thus, the unit expression

*The function  
AtomicUnit*

```
AnIdentifier.Unit / AtomicUnit(AnIdentifier.Unit)
```

will result in a (unitless) unit value that exactly represents the scale factor between the unit of an identifier and its associated atomic unit expression. You can obtain the corresponding numerical value, to be used in numerical expressions, by applying the function `EvaluateUnit` discussed in the next section.

The function `ConvertUnit` returns the unit value corresponding to the unit expression of the first argument, but taking into consideration the convention specified in the second argument. If the first argument contains a reference to a `.Unit` suffix, AIMMS will apply the full range of conversions including those specified in the `PER IDENTIFIER` attribute of the convention.

*The function  
ConvertUnit*

The expression

*Examples*

```
ConvertUnit(AnIdentifier.Unit, ConventionUsed)
```

returns the associated unit of the identifier `AnIdentifier` as if the convention `ConventionUsed` were active. A further example of the use of the function `ConvertUnit` is given in Section 30.5.1.

---

### 30.6.2 Converting unit expressions to numerical expressions

Although numerical values and unit values are two very distinct data types in AIMMS, the distinction between the two in real life applications is not always as strict. For instance, in the previous section the computation of the ratio between a unit and its associated atomic unit expression returned a unit value, which represents nothing more than a (unitless) scale factor. In practice, however, it is the numeric scale factor value that is of interest, and can be used in numerical computations.

*Numeric value  
of a unit  
expression*

Using the function `EvaluateUnit` you can compute the numerical value associated with a computed unit expression. Its syntax is:

*The function  
EvaluateUnit*

- `EvaluateUnit(computed-unit-expression)`

The numeric function value precisely corresponds to one unit of the specified computed unit expression, measured in the evaluated unit of its argument.

The following assignment to the scalar parameter `ScaleFactor` computes the (unitless) scale factor between the unit of an identifier and its associated atomic unit expression.

*Example*

```
ScaleFactor := EvaluateUnit( AnIdentifier.Unit / AtomicUnit(AnIdentifier.Unit) );
```

As you will see in the next section, the function `EvaluateUnit` offers extension the local unit override capability. The argument of `EvaluateUnit` can be a computed unit expression (see Section 30.6), whereas local unit overrides can only accept simple unit expressions.

*Extension of  
local overrides*

---

## 30.7 Locally overriding units

In some rare occasions the unit specified in the declaration of a particular identifier does not necessarily have to match with the unit of the data for that identifier. In that case, AIMMS allows you just to override the unit of a particular expression locally. Such a local unit override of an expression always takes the simple form

*Locally  
overriding units*

```
(expression) [simple-unit-expression]
```

where *expression* is some AIMMS expression, and *simple-unit-expression* is a simple unit expression as explained in Section 30.6. If *expression* solely consists of a numeric constant, AIMMS allows you to omit the parentheses around it.

You can use local unit overrides in a variety of data I/O related situations.

*Where to use*

- In a WRITE, DISPLAY or PUT statement, you can use a local unit override to specify the particular unit in which data must be written to a file, database table or window.
- In the FormatString function, you can use a local unit override to specify the unit in which a numeric argument corresponding to a %n format specifier must be formatted.
- On the left side of a data assignment, in the header of a composite table, or in a READ statement, you can use a local unit override to specify the unit in which the supplied data to be provided.

In all these data I/O statements and expressions, AIMMS requires that the unit provided in the override is commensurate with the original unit that can be associated with the expression.

*Commensurate requirement*

Given the declarations of the examples in the Section 30.4, the following data I/O statements locally override the default unit [km/h] of the identifier VelocityOfItem with the commensurate unit [mph].

*Example*

- Override per identifier:

```
(VelocityOfItem) [mph] := DATA { car: 55, truck: 45 };
read (VelocityOfItem) [mph] from table VelocityTable;
display (VelocityOfItem) [mph];
```

- Override per individual entry:

```
put (VelocityOfItem('car')) [mph];
StringVal := FormatString("Speed in [mph]: %n", (VelocityOfItem('car')) [mph]);
```

Recall that parentheses are always required when you want to override the default unit in expressions and statements, unless the overridden expression is a simple numeric constant.

In addition to overriding units during a data exchange, you can also override the unit of a (sub)expression in an assignment with the purpose of enforcing unit consistency of all terms in the assignment. This is especially useful when there are numeric constants inside your expressions. AIMMS will add the appropriate scale factor if the specified unit override does not match with the corresponding atomic unit expression.

*Override for consistency*

The following examples illustrate unit overrides with the purpose of enforcing unit consistency. *Examples*

- Consider the assignment

```
SoundIntensity := (10 * log10( SoundLevel / ReferenceLevel )) [dB];
```

If `SoundIntensity` has an associated unit of [dB], the right hand side of the assignment, which by itself is unitless, must be locally overridden to make the entire assignment unit consistent.

- Consider the assignment

```
a := b + 10 [km];
```

where both `a` and `b` are measured in terms of length. As discussed in Section 30.4, AIMMS will make no assumption about the unit associated with the numerical constant 10 in the expression on the right-hand side of the assignment. In order to make the assignment unit consistent, an explicit unit override of the constant term is required. If the associated base unit is [m], AIMMS will automatically add a scale factor of 1000, whence the assignment will numerically evaluate to `a := b + 10*1000`.

If you explicitly associate a unit with an expression which already contains one or more identifiers *with* associated units, the numerical result can be unexpected. This is due to fact that AIMMS, during expression evaluation, uses the unscaled numerical values with respect to the associated atomic units of each identifier. To illustrate, reconsider the assignment

*Caution is needed*

```
a := (b * c) [km];
```

but now assume that the identifiers `a`, `b`, and `c` have units [km], [km], and [10\*m]. If the values of `b` and `c` are 1 [km](=1000 [m]) and 50 [10\*m](=500 [m]), respectively, the numerical result of `a` after the assignment will amount to  $(500 * 1000) * 1000$  [m]= 500000 [km], which may not be the result that you intended.

---

## 30.8 Globally overriding units through CONVENTIONS

In addition to locally overriding the unit definition of an identifier in a particular statement, you can also *globally* override the default format for data exchange using `READ` and `WRITE`, `DISPLAY` and `SOLVE` statements by selecting an appropriate *unit convention*. A convention offers a global medium to specify alternative (scaled) units for multiple quantities, units, and identifiers. In addition, one can specify alternative representations for a calendar in a convention.

*Unit conventions*

Once you have selected a convention, AIMMS will interpret all data transfer with an external component according to the units that are specified in the convention. When no convention has been selected for a particular external component, AIMMS will use the default convention, i.e. apply the unit as specified in the declaration of an identifier. For a compound quantity not present in a convention, AIMMS will apply the convention to all composing atomic units used in the compound quantity.

*Effect of conventions*

Conventions must be declared before their use. The list of attributes of a CONVENTION declaration are described in Table 30.5.

*Convention attributes*

Attribute	Value-type	See also page
TEXT	<i>string</i>	
COMMENT	<i>comment string</i>	
PER IDENTIFIER	<i>convention-list/reference</i>	
PER QUANTITY	<i>convention-list</i>	
PER UNIT	<i>convention-list</i>	
TIMESLOT FORMAT	<i>timeslot-format-list</i>	498

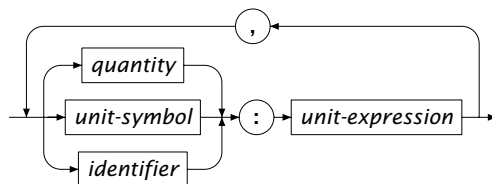
Table 30.5: Simple CONVENTION attributes

A convention list is a simple list associating single quantities, units and identifiers with a particular (scaled) unit expression. The specified unit expressions must be consistent with the base unit of the quantity, the specified unit, or the identifier unit, respectively.

*Convention list*

*convention-list :*

*Syntax*



In addition to a fixed convention list, the PER IDENTIFIER attribute also accepts a reference to a unit-valued parameter defined over the set AllIdentifiers or a subset thereof. In that case, the convention will dynamically construct a convention list based on the contents of the unit-valued parameter.

*Customizable conventions*

The following declaration illustrates the use of a CONVENTION to define the more common units in the Anglo-American unit system at the quantity level, the unit level and the identifier level.

*Example*

```
CONVENTION:
  identifier      : AngloAmericanUnits
  per identifier  : GasolinePurchase : gallon,
                  PersonalHeight   : feet
  per quantity   : Velocity        : mph,
                  Temperature      : degF,
                  Length           : mile
  per unit       : cm : inch,
                  m  : yard,
                  km : mile ;
```

Assuming that IdentifierUnits is a unit-valued parameter defined over All-Identifiers, the following CONVENTION declaration illustrates a convention that can be customized at runtime by modifying the contents of the unit parameter IdentifierUnits.

*Customizable example*

```
CONVENTION:
  identifier      : CustomizableConvention
  per identifier  : IdentifierUnits ;
```

For a particular identifier, AIMMS will select a unit from a convention in the following order.

*Application order*

- If a unit has been specified for the identifier, AIMMS will use it.
- If the identifier can be associated with a specific quantity in the convention, AIMMS will use the unit specified for that quantity.
- In all other cases AIMMS will apply the convention to an atomic unit directly, or to all composing atomic units used in a compound unit.

In addition to globally overriding units, CONVENTIONS can also be used, through the TIMESLOT FORMAT attribute, to override the time slot format of calendars. You may need to specify alternative time slot formats, for instance, when you are reading data from an external database or file, in which all dates are not specified in the same time zone as the one your model assumes. The TIMESLOT FORMAT attribute of a CONVENTION is discussed in full detail in Section 31.10.

*Timeslot format list*

You can declare more than one convention in your model. A CONVENTION attribute can be specified for the following node types in the model tree, which all correspond to an external component:

*The CONVENTION attribute*

- the main model (used for the end-user interface or as default for all other external components),
- a mathematical program,

- a file (also when used to refer to a DLL containing a library of external procedures and functions used by AIMMS), and
- a database table or procedure.

The value of the CONVENTION attribute can be a specific convention declared in your model, or a string or element parameter referring to a particular unit convention.

For data exchange with all aforementioned external components AIMMS will select a unit convention in the following order.

*Convention semantics*

- If an external component has a nonempty CONVENTION attribute, AIMMS will use that convention.
- For display in the user interface, or for data exchange with external components without a CONVENTION attribute, AIMMS will use the convention specified for the main model (see also Section 33.2), if present.
- If the main model and external components have no CONVENTION attribute, AIMMS will use the default convention, i.e. use the unit as specified in the declaration of each identifier.

The following declaration of a FILE identifier illustrates the use of the CONVENTION attribute. All the output to the file ResultFile will be displayed in Anglo-American units.

*Example*

```
FILE:
  identifier : ResultFile
  name      : "Output\\result.dat"
  convention : AngloAmericanUnits ;
```

---

## 30.9 Unit-valued parameters

In some cases not all entries of an indexed identifier have the same associated unit. An example is the diet model where the nutritive value of each nutrient for a single serving of a particular food type is measured in a different unit.

*Parametrized units*

In order to deal with such situations, AIMMS allows the declaration of (indexed) *unit-valued* parameters which you can use in the unit definition of the other parameters and variables in your model. In the model tree, unit-valued parameters are available as a special type of parameter declaration, with attributes as given in Table 30.6.

*Unit-valued parameters*

Attribute	Value-type	See also page
INDEX DOMAIN	<i>index-domain</i>	
QUANTITY	<i>quantity</i>	
DEFAULT	<i>unit-expression</i>	
PROPERTY	NoSave	45
TEXT	<i>string</i>	19
COMMENT	<i>comment string</i>	19, 31
DEFINITION	<i>unit-expression</i>	

Table 30.6: UNIT PARAMETER attributes

You should specify the QUANTITY attribute if all unit values stored in the unit parameter can be associated with a single quantity declared in your model. The effect of specifying a quantity in the QUANTITY attribute of a unit parameter is twofold:

*The QUANTITY attribute*

- during assignments to the unit parameter, AIMMS will verify whether the assigned unit values are commensurate with the base unit of specified quantity, and
- AIMMS will modify its (compile-time) unit analysis to use the specified quantity rather than an artificial quantity based on the name of the unit parameter (see below).

The DEFAULT and DEFINITION attributes of a unit parameter have the same purpose as the DEFAULT and DEFINITION attribute of ordinary parameters, except that the resulting values must be unit expressions (see Section 30.6). If you have specified a quantity in the QUANTITY attribute, AIMMS will verify that these unit expressions are commensurate with the specified quantity.

*The DEFAULT and DEFINITION attributes*

All unit values read from an external data source, or assigned to a unit parameter, either via an assignment or through its DEFINITION attribute, must evaluate to existing unit symbols only. A compile- or runtime error will occur, when a unit value refers to a unit symbol that is not defined in any of the QUANTITY declarations contained in your model.

*Allowed unit values*

With unit parameters you can create, store and manipulate scalar or multidimensional collections of unit values. The unit values stored in a unit parameter can be used, for instance:

*Use of unit parameters*

- to associate a parametrized (i.e. multidimensional) collection of units with a single multidimensional identifier (through its UNIT attribute), or
- to specify a local unit override based on a unit (or collection of units) that is not known a priori.

When a UNIT attribute of an identifier contains a reference to a unit parameter, this can, but need not, modify the way in which AIMMS conducts its usual unit analysis. There are two distinct scenarios, both described below.

*Unit analysis...*

If the unit parameter has an associated quantity (specified through its QUANTITY attribute), all units stored in the unit parameter are known to be commensurate with the base unit of the quantity, although the individual scale factors may be different if the unit parameter is multidimensional. In this case, AIMMS will base its unit analysis on the associated quantity.

*... with associated quantity*

If there is no associated quantity, AIMMS will introduce an artificial quantity solely on the basis of the symbolic name of the unit parameter (i.e. without consideration of its dimension), and base all further unit analysis on this artificial quantity only. If there is unit consistency at the level of these artificial quantities, this automatically ensures, for multidimensional unit parameters, unit consistency at the individual level as well, regardless of the specific individual unit values stored in it.

*... without associated quantity*

Consider the following declarations of unit-valued parameters, where *f* is an index into the set *Foods* and *n* an index into the set *Nutrients*.

*Example*

```
UNIT PARAMETER:
  identifier : NutrientUnit
  index domain : n ;
UNIT PARAMETER:
  identifier : FoodUnit
  index domain : f ;
```

With these unit-valued parameters you can specify meaningful indexed unit expressions for the UNIT attribute of the following parameters.

```
PARAMETER:
  identifier : NutritiveValue
  index domain : (f,n)
  unit : NutrientUnit(n)/FoodUnit(f) ;
PARAMETER:
  identifier : NutrientMinimum
  index domain : n
  unit : NutrientUnit(n) ;
VARIABLE:
  identifier : Serving
  index domain : f
  unit : FoodUnit(f) ;
```

With these declarations, you can now easily verify that all terms in the definition of the following constraint are unit consistent at the symbolic level.

```
CONSTRAINT:
  identifier : NutrientRequirement
```

```

index domain : n
unit          : NutrientUnit(n)
definition    : sum[ f, Servings(f)*NutritiveValue(f,n) ] >= NutrientMinimum(n) ;

```

When the UNIT attribute of an identifier is parametrized by means of indexed unit parameter, AIMMS will correctly scale all data exchange with external components (see Section 30.5). During data exchange with an external component, AIMMS considers the specified units at the individual (indexed) level, and will determine the proper scaling for every individual index position. In addition, when a unit convention is active, AIMMS will scale all individual entries according to that convention, as applied to the corresponding individual entries of the indexed unit parameter. As usual, all data of an identifier with a parametrized associated unit will be stored internally in the corresponding atomic unit of every individual index value.

*Indexed scaling*

When AIMMS generates mathematical program which contains the variable `Serving(f)`, each column corresponding to this variable will be scaled according to the scale factor of the particular unit stored in `FoodUnit(f)` with respect to their corresponding atomic unit expressions. Similarly, AIMMS will scale the columns corresponding to the constraint `NutrientRequirement(n)` according the scale factors of the units stored in `NutrientUnit(n)` with respect to their corresponding atomic unit expressions.

*Example revisited*

You can initialize a unit-valued parameter through lists, tables, and composite tables like you can initialize any other AIMMS parameter (see Chapter 26). The values of the individual entries must be valid unit constants (see Section 30.6), and must be surrounded by square brackets. For compound units constants you can optionally indicate the associated quantity in a similar way as in the unit definition of a parameter.

*Initializing unit-valued parameters*

The following list initializes the unit-valued parameter `NutrientUnit` for a particular set of Nutrients.

*Example*

```

NutrientUnit := DATA { Energy : [kJ] ,
                        Protein : [mg] ,
                        Iron   : [%RDA] };

```

In addition, AIMMS allows you to read the initial data of a unit parameter from a database table, and write the values of a unit parameter to a database table. The unit values in the database table must be unit constants, and must be stored without square brackets.

*Unit parameters and databases*

When a composite table in a data file, or a table in a database contains both the values of a multidimensional unit parameter, and a corresponding numeric parameter whose UNIT attribute references that unit parameter, AIMMS allows you to read both identifiers in a single pass. When reading both identifiers, AIMMS will make sure that the numeric values are interpreted with respect to the corresponding unit value that is read simultaneously.

*Simultaneous unit and data initialization*

AIMMS even allows you to make assignments from identifiers with a constant unit to identifier slices of identifiers with a parametrized unit and vice versa. If AIMMS detects this special situation during compilation of your model, it will postpone the compile unit consistency check whenever necessary, and replace it with a runtime consistency check which is performed every time the assignment is executed. Because all data is stored by AIMMS with respect to atomic units internally, unit consistency again automatically implies scale consistency.

*Constant versus parametrized units*

Given the declarations of the previous example, assume the existence of an additional parameter `EnergyContent(f)` with a constant associated unit, say `Kcal`. Then, AIMMS will postpone the compile unit consistency check for the following two statements, and replace it with a runtime check.

*Example*

```
NutritiveValue(f,'Energy') := EnergyContent(f);
EnergyContent(f)           := NutritiveValue(f,'Energy');
```

The runtime unit consistency check will only succeed, whenever the unit value of the unit parameter `NutrientUnit('Energy')` is commensurate with the constant unit `Kcal`.

AIMMS will only replace a compile time with a runtime unit consistency check if a unique unit can be associated with the right-hand side of the assignment at compile time. If the assigned expression consists of subexpressions which have different associated unit expressions at compile time, a compile time error will result. This is even the case when, at runtime, these unit expressions evaluate to units that are commensurate with the unit of the left-hand side of the assignment.

*Restrictions*