TMS Physics Delphi Development Library

DEVELOPERS GUIDE

May 2018

Web: http://www.tmssoftware.com
Email: info@tmssoftware.com

Index

Introduction

PHYSICS is a Delphi library for developers. PHYSICS library contains special classes and allows using various physics concepts (such as physical quantities, units of measurement and so on) in Delphi programs.

ADVANTAGES of PHYSICS library:
1. 100% Delphi code.
2. Strongly structured class hierarchy (as close as possible to modern physics concepts).
3. Universal algorithms for working with physics concepts (no need to modify primary code).
4. Many predefined physical entities (physical quantities, units of measurement).
5. Easy to introduce new physics concepts (physical quantities, units of measurement and so on).
6. Platform independent (many platforms supported – x32, x64, VCL, FMX).

Dependences

PHYSICS Delphi library depends on:
1. RTL.
2. DRTE (Delphi Run-Time Environment) library.

Extensions

There are the following extensions of PHYSICS Delphi library:
1. Physics Measurement.
2. Mathphysics (depends on MATHEMATICS library).

Basic concepts

The main goal of PHYSICS library is to provide the easiest way for converting values between various units of measurement. This part explains main concepts used in PHYSICS library.

Physical quantities

A **physical quantity** is a property of a physical substance that can be measured (http://en.wikipedia.org/wiki/Physical_quantity). Examples of physical quantities are: mass, time, electric current and so on. One should not confuse the concept of physical quantity and the concept of physical (quantity) values. A physical quantity is an abstract concept, and a physical quantity value is the value of concrete phenomenon property. For example, 5 grams is a physical value of the physical quantity mass.

There are fundamental (base) physical quantities called dimensions: length, time, mass, temperature, electric current, amount of substance, luminous intensity. They are fundamental because all other quantities can be expressed via them. For example, the physical quantity velocity can be expressed via the fundamental quantities length and time: **velocity = length/time**. Therefore, any physical quantity has the same characteristic called physical dimension. The velocity quantity has the dimension **length/time**, the acceleration quantity has the dimension **length/time^2**, the area quantity has the dimension **length^2** and so on.

Thus, any physical quantity can be characterized by two properties: name (or symbol) and dimension. There are fundamental quantities with the same dimensions and derived quantities with dimensions, constructed from the fundamental quantities.

NOTE. Three additional dimensions can be introduced for convenience: **information**, **plane angle**, **solid angle**. These quantities are dimensionless in fact, but they can be considered as the additional dimensions to differentiate them from truly dimensionless data (numbers).

The total list of dimensions and physical quantities, defined in PHYSICS library, can be found in **Appendix A**.

Units of measurement

An unit (of measurement) is a definite magnitude of a physical quantity (http://en.wikipedia.org/wiki/Unit_of_measurement). For, example, meter is a predefined magnitude of the length physical quantity. Thus, as a physical quantity, every unit has such attribute as physical dimension. Some unit can measure a physical quantity **if and only if** they have **equal** dimensions. There is 'many to many' relation between physical quantities and units. That is, many units can measure one quantity, and one unit can measure many quantities. For

example, mass can be measured with grams, pounds, kilograms and so on. At the same time, one unit – Pascal, can measure hydrostatic pressure and mechanical stress.

In accordance with said above, there are base units to measure fundamental quantities. For the length quantity – meters, inches and so on, for the mass quantity – grams, pounds and so on… All units for the derived quantities can be produced by multiplying and dividing the base units, multiplying them by some constants and adding constant values to them (using general algebra rules). For example, the unit of the area measurement 'Are' can be got multiplying **10 m** by **10 m**, so **1 a = 100 m^2**.

One of the main purposes to use units of measurement in a computer program is to convert values of physical quantities from one unit to other ones. The base units are converted in accordance with their definitions. For example, to convert a temperature value t from degrees of Celsius to degrees of Fahrenheit the following equation is used **t °C = 5/9 (t °F - 32)**. All the base units are 'linear' because they have 'one dimension'. So, they all converted from one to another by a linear law. The law of the derived unit conversion is determined by the way of their derivation. For example, **1 m^2 = 1 m · 1 m = 100 cm · 100 cm = 10000 cm^2**.

Even the algorithm of the conversion is straightforward for any separate case of units, it is not so easy to write the code of the algorithm in general case. One of the main difficulties is to create 'right' class hierarchy that provides one algorithm for all units and for that time an easy way to introduce new units.

PHYSICS library has **unique universal algorithm** of unit conversion. It converts value **from one unit to any other unit** if they are compatible for the conversion (have equal dimensions). The only thing needed is to create new unit class. The algorithm itself will 'build' the conversion function to any other compatible unit, according to the unit definition.

Unit prefixes

A unit prefix is a specifier that indicates multiplication factor for the unit it precedes. A simple example of a prefix using is the definition of the kilogram unit **kg**. Here the prefix kilo '**k**' precedes the unit gram '**g**'. It means that the magnitude of the unit must be multiplied with the value of the prefix, in this case it is **1000**.
PHYSICS library supports many decimal and binary prefixes (see **Appendix B**).

Using PHYSICS

The main functionality of PHYSICS library is conversion of different units of measurement. Another functionality is to convert string data to units of measurement. The functionality is

mainly realized with the ***Unit***[1] class and the ***UnitConverter***. The next part explains how to use the classes.

Unit conversion

The unit conversion is very simple to implement with PHYSICS library. The base class ***Unit*** has static method ***Convert*** to convert value from any unit to any compatible one. The following code demonstrates an example of the conversion

```
var
  u1: TUnit;
  u2: TUnit;
  x1, x2: TFloat;
begin
  u1:=TPascal.Create;
  u2:=TAtmosphere.Create;
  x1:=100;
  x2:=TUnit.Convert(u1, u2, x1);
  …
```

In this example, a real value converted from the Pascal unit to the Atmosphere unit. Units must be compatible to convert values. Units are compatible if they have equal dimensions. Use the ***Unit*** class method ***Convertible*** to check that two units are compatible for conversion.

Fast unit conversion

The unit conversion, described above, is universal and can be applied to any (convertible) units. But it can be slow for conversion of huge arrays of values. The more 'complicated' units are the slower conversion is.

At the same time, all units can be divided into two categories: 'zero based' and not 'zero based'[2]. The 'zero based' units are such units that their scale zeros equal. For example, mass units are zero based, because zero mass is always zero, no matter what unit is used to measure it. As opposite, temperature units are not zero based, because **0°C is not equal to 0°K**.

The conversion of the 'zero based' units if always linear and can be made using one factor coefficient **f**. Then the formula is $\mathbf{x\ unit_1 = f \cdot x\ unit_2}$, where **x** is the value, $\mathbf{unit_1}$ and $\mathbf{unit_2}$ are the units to convert from and to convert to, **f** is the conversion factor. The ***Unit*** class contains the ***IsZeroBased*** method to check, if a unit is zero based.

---

1 Hereinafter in the text (not code) prefix 'T' is omitted for all class names.

2 For other cases, see 'Special units' part.

The conversion of the not 'zero based' units cannot be implemented by using only one scale factor in general case. But, in the case of 'interval' conversion, this can be made by multiplying the value with a 'scale factor'. For example, the temperature interval of **10°C** can be converted to the temperature interval measured in Fahrenheit's degrees by multiplying the first with **9/5**.

In accordance with said above, for all 'zero based' units and for 'interval' conversion a scale factor can be used to convert values. The ***Unit*** class contains the static method ***ScaleFactor*** to calculate the value of the scale factor. The example code below shows how to use this method to convert an array of values.

Let there is an array of pressure values measured in the 'millimeters of mercury'.

```
var
   pressures: TArray<TFloat>;
```

And all values must be converted to the Pascal units. The code of the conversion is the following:

```
var
  u1, u2: TUnit;
  v: TFloat;
  i: Integer;
  …
begin
  …
  u1:=TMillimetreOfMercury.Create;
  u2:=TPascal.Create;
  try
    v:=TUnit.ScaleFactor(u1, u2);
    for i:=0 to Length(pressures)-1 do
      pressures[i]:=v*pressures[i];
  finally
    u1.Free;
    u2.Free;
  end;
end;
```

String to Unit conversion

PHYSICS library provides mechanisms to convert string data to units of measurement. It is useful for all programs which allow the user to input units to measure some value. The

mechanism of string-to-unit conversion is provided by the ***UnitConverter*** class. The following code demonstrates conversion of a string to a unit.

```
var
  u: TUnit;
  uconverter: TUnitConverter;
  s, error: string;
begin
  uconverter:=TUnitConverter.Create;
  s:='N mm^2/ns';
  u:=uconverter.ConvertString(s,error);
  …
```

In this example new unit x is constructed from string and it is **Newton·millimeter^2/nanosecond**.

The rules of string-to-unit conversion:

- A string can contain unit symbols, prefixes, spaces, division operator sign '/' and power operator sign '^'.
- No multiplication sign allowed in a string (space symbol ' ' used instead).
- Only one division sign allowed in a string.
- A prefix is not separated by spaces with the unit it belongs to.
- Different units are separated by spaces (that is spaces used as multiplication operator).

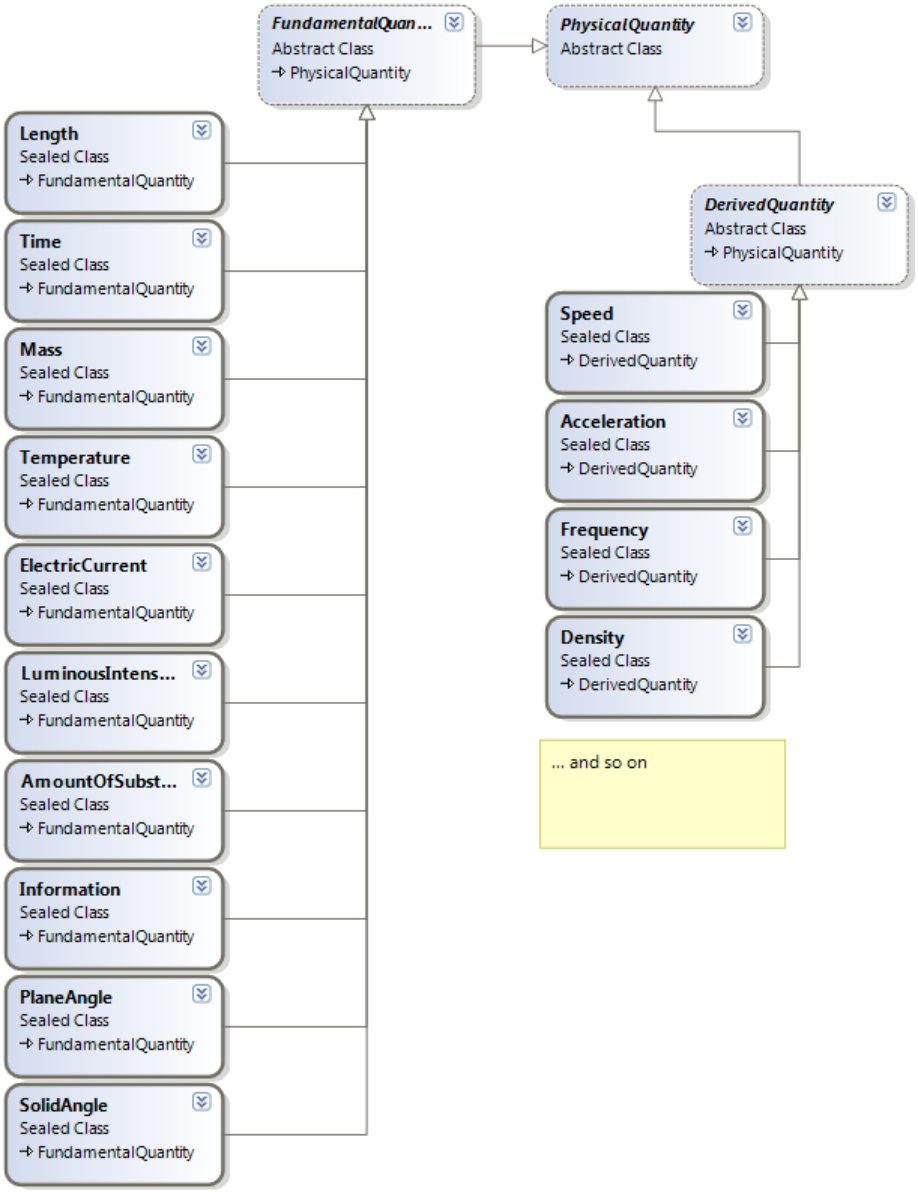
If there is an error in the string, its description is returned via error out parameter.
The list of all defined named units can be found in **Appendix B**.

Class Hierarchy

This part explains the base class hierarchy of PHYSICS library. There is no need to know the hierarchy for using main features of the library: unit conversion and parsing string data into units of measurement. The knowledge is only useful for extending the library - introducing new physical quantities and units of measurement.

Physical quantity class hierarchy

In accordance with said above, the class hierarchy of physical quantities in PHYSICS library follows the concepts of modern physics. Each physical quantity is represented by a class. Base abstract class for all physical quantities is the ***PhysicalQuantity***. It has three main properties: Name, Symbol and Dimension. The ***FundamentalQuantity*** and ***DerivedQuantity*** classes are directly inherited from this base class. Ten fundamental quantities (***Length***, ***Time***, ***Mass***, ***Temperature***, ***ElectricCurrent***, ***AmountOfSubstance***, ***LuminousIntensity***, ***Information***, ***PlaneAngle***, ***SolidAngle***) are inherited from the former class. All other derived quantity classes are inherited from the last one. The class hierarchy diagram is shown on picture 1.1.

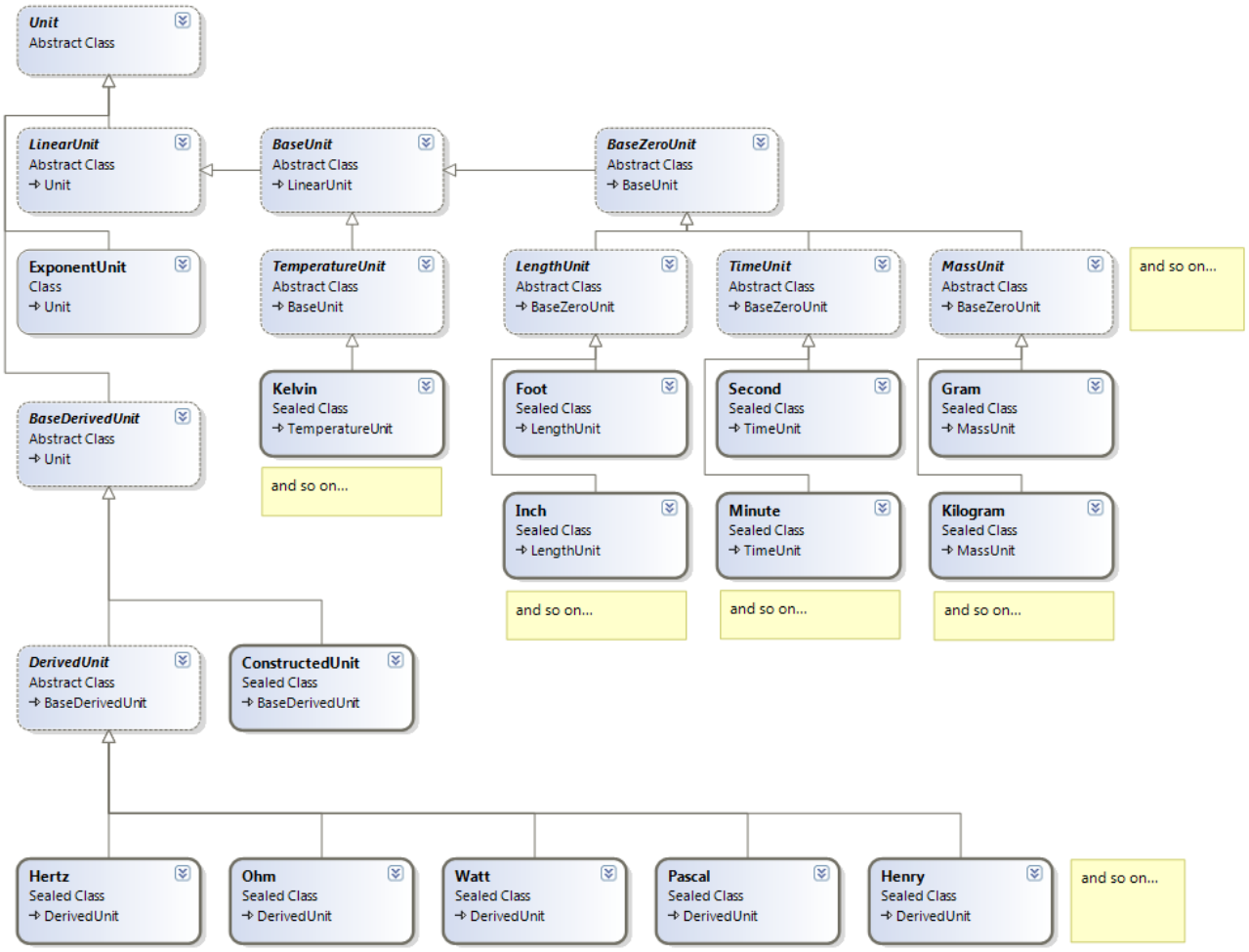


Picture 1.1. Physical quantity class hierarchy diagram.

Unit class hierarchy

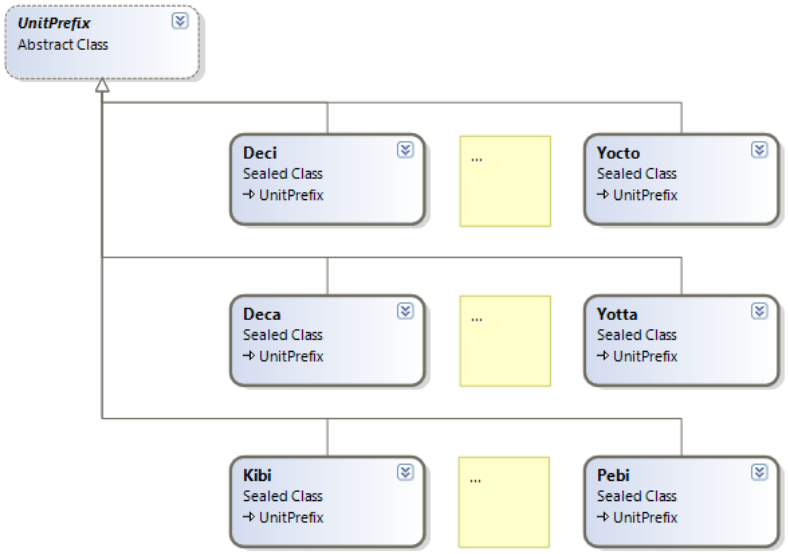
The unit class hierarchy is shown on picture 1.2. The hierarchy is rather complicated. The hierarchy was constructed with the purpose to represent the modern physics concept of 'units of measurement' as close as possible and, from the other side, to provide an universal algorithm of unit conversion and the easiest way to introduce new units.

The base abstract class for all units is ***Unit***. It provides all common unit properties – Name, Symbol, Dimension, and common algorithms – conversion values from one unit to another, checking units for compatibility and so on.



Picture 1.2. Unit class hierarchy diagram.

The ***LinearUnit*** class is directly derived from the base abstract class ***Unit***. It realizes the abstract methods for the linear conversion algorithm. Then, the ***BaseUnit*** class is inherited from the ***LinearUnit*** class. It realizes the abstract mechanisms for all base units (those measure the fundamental quantities). The base units are separated into eleven branches, corresponding to the fundamental quantities. All other units, corresponding the fundamental quantities, must be inherited from one of these eleven units. There are many predefined base units: ***Foot***, ***Gram***, ***Second*** and so on.



Picture 1.3. Unit prefix class hierarchy diagram.

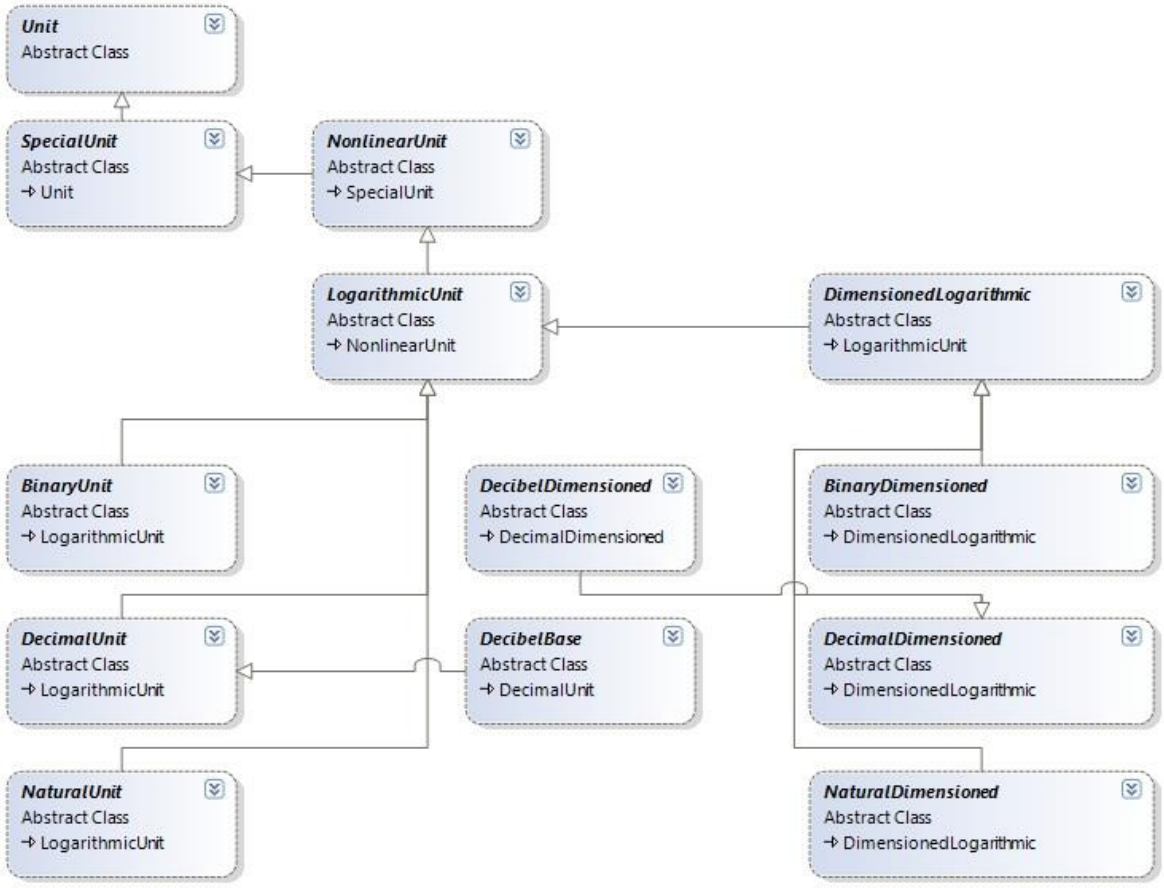
All other units are constructed from the base ones. The simplest way to get new unit is to multiply it by some value and/or exponentiate it. For example, from the meter unit **m** new unit can be constructed **10 m^2**. The ***ExponentUnit*** class realizes this concept. The special abstract class ***UnitPrefix*** (picture 1.3) is designed to realize standard multipliers for units (http://en.wikipedia.org/wiki/SI_prefix). All standard multipliers realized in PHYSICS library and can be used for unit derivation.

And finally, the general way to derive new unit from the base ones is to multiply several exponent units. For example, the following unit can be constructed: **kg·m^2·A^{-1}·s^{-2}**. This concept is realized by the ***BaseDerivedUnit*** class. All derived units can be divided in two categories. The first category includes the units those have their own names. For example, Newton, Pascal, Henry and so on. The second category includes all other units, they have no names, only symbols. For example **km·s^{-2}**. These concepts realized by the ***DerivedUnit*** and ***ConstructedUnit*** classes correspondingly.

PHYSICS library implements many predefined units of measurement. New units can be simply added to the library without modifying the core algorithms.

Special units (logarithmic and decibels)

The unit class hierarchy presented above is not full. It was reduced for the reason of simplification. Really, there is another branch of inheritance shown on picture 1.4. The main branch node is ***SpecialUnit***. This branch was introduced for special needs. The ***SpecialUnit*** class used in all core algorithms of the library, but implementation supposes special cases.



Picture 1.4. Diagram of special unit class branch of inheritance.

Common example of special unit is Decibel – this is a logarithmic unit with special rules of conversion and operations (https://en.wikipedia.org/wiki/Decibel). The Decibel is defined as logarithmic unit by base 10, so it inherited from Decimal logarithmic unit. Really, all logarithmic units are dimensionless, because they express the relative values. However, dimensioned decibels can be introduced by supposing some dimensioned reference value (https://en.wikipedia.org/wiki/Decibel#Suffixes_and_reference_values).

Common feature of the logarithmic (and decibel) units is that they are NOT SCALABLE. It means that 'interval' conversion for them cannot be done by multiplying with a factor value (in general case). The conversion between logarithmic units and other ones can be done only with general conversion algorithm (no fast conversion). Use method ***IsScalable*** of the ***Unit*** class for checking if unit can be converted to another one by the scale factor. Also, method ***MakeConversion*** can be used for creating suitable conversion delegate for any units, taking into account their properties.

Summary of logarithmic and decibel unit features:

- Logarithmic units are not scalable with other ones.
- There are dimensionless and dimensioned decibels.
- Logarithmic units and decibels cannot be multiplied with other logarithmic ones.

- Logarithmic units and decibels cannot be raise to a power (except 1).
- When making operations with physical values logarithmic units make special sense: the logarithmic values cannot be multiplied; the logarithmic values cannot be raise to a power; sum of two logarithmic values can be done if they have the same dimension or if one of them is dimensionless.

Extending PHYSICS

PHYSICS library provides a complete core for converting units of different types and to parse string data into units of measurement. It also contains many predefined, completely realized, physical entities. Thus, the library can be used without any modification for its main purposes.
Nevertheless, there is a huge amount of specific physical entities used in many specialized applications. PHYSICS library provides the easiest way to introduce new specific physical entities. They will be used like other predefined entities without any modification of the core algorithms.

Introducing new physical quantity

It is very easy to introduce new physical quantity. Every quantity is represented as a class inside the program. Thus, new class must be inherited from the ***DerivedQuantity*** class. For example, to introduce the force physical quantity the following class must be created:

```
interface
…
  TForce = class sealed (TDerivedQuantity)
  protected
    function GetSymbol: string; override; final;
    function GetName: string; override; final;
    function GetDimension: TDimension; override; final;
  end;
…
implementation
…
function TForce.GetDimension: TDimension;
begin
  Result:=(Mass*Length)/(Time*Time);
end;

function TForce.GetName: string;
begin
  Result:='Force';
end;
```

```
function TForce.GetSymbol: string;
begin
  Result:='F';
end;
…
```

This code introduce new physical quantity with the name ‘Force’ and the symbol ‘F’. The dimension of the quantity is **mass·length/time^2** (which corresponds the force dimension).

And it is all. PHYSICS library uses Delphi reflection mechanisms to find and identify physics entities. New class will be automatically found and registered. All algorithms of working with physical quantities do not depend on the type of the quantity. The program can find quantity, display it, find units of measurement those are compatible with the quantity and so on.

Introducing new units of measurement

The unit class hierarchy is rather complicated, but there is no need to know all features of the structure to introduce new units in a program and use them. PHYSICS library provides simple mechanisms to introduce and use new units.

There are two different ways to use new units in a program. The first is to create new classes for the units. This way must be used for all units, those have their own names (Pascal, Newton and so on). For such units, corresponding the base physical quantities, the class must be inherited from one of the base derived units. For example, there is the code for the ‘yard’ unit.

```
interface
…
  // Yard = 0.9144 m
  TYard = class sealed (TLengthUnit)
  protected
    function GetFactor: TFloat; override; final;
    function GetName: string;   override; final;
    function GetPlural: string; override; final;
    function GetSymbol: string; override; final;
  end;
…
implementation
…
function TYard.GetFactor: TFloat;
begin
```

```
  result:=0.9144;
end;

function TYard.GetName: string;
begin
  result:='Yard';
end;

function TYard.GetPlural: string;
begin
  result:='Yards';
end;

function TYard.GetSymbol: string;
begin
  result:='yd';
end;
…
```

The ***Yard*** unit is inherited from the ***LengthUnit***, because it is a measure for the **length** physical dimension. It overrides the methods providing the name and the symbol of the unit. Also it overrides the ***GetFactor*** method, which returns the coefficient of conversion to the base unit. It is the only information needed for the conversion algorithm. All coefficients must correspond to conversion of the unit to the base SI units (http://en.wikipedia.org/wiki/SI).

All complex units must be directly inherited from the ***DerivedUnit*** class. For example, there is the code for the 'watt' unit.

```
interface
…
  // Watt ( kg m^2/s^3 )
  TWatt = class sealed (TDerivedUnit)
  protected
    function GetName: string;   override; final;
    function GetPlural: string; override; final;
    function GetSymbol: string; override; final;
    procedure RecreateUnits; override; final;
  end;
…
implementation
…
function TWatt.GetName: string;
```

```
begin
  result:='Watt';
end;

function TWatt.GetPlural: string;
begin
  result:='Watts';
end;

function TWatt.GetSymbol: string;
begin
  result:='W';
end;

procedure TWatt.RecreateUnits;
begin
  inherited;
  SetLength(fUnits, 3);
  fUnits[0]:=TExponentUnit.Create(TKilogram.Create, 1);
  fUnits[1]:=TExponentUnit.Create(TMetre.Create, 2);
  fUnits[2]:=TExponentUnit.Create(TSecond.Create, -3);
end;
…
```

The class overrides the methods providing the name and symbol of the unit. Also it overrides the ***RecreateUnits*** method which provides main information for conversion and other algorithms. In this method the unit 'watt' is constructed with three exponent units **Kilogram^{1}·Metre^{2}·Second^{-3}.**

The second way of using new units in a program is creating object instances dynamically in run-time. The simplest way is using the ***ConstructedUnit*** class. The following code demonstrates run-time unit creation:

```
var
  x: TConstructedUnit;
  u: TExponentUnit;
begin
  x:=TConstructedUnit.Create;
  u:=TExponentUnit.Create(TKilogram.Create,1);
  x.AddUnit(u);
  u:=TExponentUnit.Create(TMetre.Create,1);
  x.AddUnit(u);
  u:=TExponentUnit.Create(TSecond.Create,-2);
```

```
  x.AddUnit(u);
  …
end;
```

The created unit x is **kg·m/s^2.** Then this constructed unit can be used as all other ones for conversion purposes.

The special case of introducing new unit of measurement is inheriting from one of the special units. Let us consider an example of implementing a dimensioned decibel unit for voltage measurement (https://en.wikipedia.org/wiki/Decibel#Voltage). The code of the class presented below:

```
interface
…
/// <summary>
/// Decibel - Voltage
/// </summary>
TDecibelVoltage = class sealed (TDecibelDimensioned20)
protected
  /// Reference unit - 1 Volt
  function CreateReferenceUnit: TUnit; override; final;
  function GetName: string;   override; final;
  function GetPlural: string; override; final;
  function GetSymbol: string; override; final;
public
  class function IsRealized: boolean; override; final;
end;
…
implementation
…
function TDecibelVoltage.CreateReferenceUnit: TUnit;
begin
  result:= TVolt.Create;
end;

function TDecibelVoltage.GetName: string;
begin
  result:= 'Decibel-Volts';
end;

function TDecibelVoltage.GetPlural: string;
begin
  result:= 'Decibel-Volts';
end;

function TDecibelVoltage.GetSymbol: string;
begin
```

```
  result:= 'dBV';
end;

class function TDecibelVoltage.IsRealized: boolean;
begin
  result:= true;
end;
…
```

As one can see from the implementation, the class overrides common methods of the ***Unit*** class, like getting unit's symbol, name etc. In addition, the class overrides ***IsRealized*** method which returns 'True' for defining that the unit provides the final functionality (unfortunately, standard Delphi RTTI system does not provide such information). As the unit is dimensioned decibel, it implements the ***CreateReferenceUnit*** method. This function returns the '*Volt*' unit, according to the unit definition. The class is inherited from the ***DecibelDimensioned20***, since the voltage decibels converted with respect to power, not amplitude, so the factor 20 used.

Note, there is no need to provide additional methods for defining that the unit is scalable or additive with other ones. The PHYSICS core algorithms detect this automatically, based on the specially designed class hierarchy.

Mathphysics extension for PHYSICS

Mathphysics extension for PHYSICS library introduces the concept of physical value: the structure containing a value (scalar, vector, tensor, …) ant a unit of measurement. This concept is useful for creating engineering applications those operate with values of different physical quantities measured with different units of measurement.

The next part contains detailed information about the physical value concept and its application.

Physical value type

The base type for a physical value is ***PhysicalValue***. This class has two virtual constructors to create value instances:

```
  constructor Create(const StrValue: string); overload; virtual;
  constructor Create(aUnit: TUnit); overload; virtual;
```

The first constructor allows creating physical value by converting a string, the second creates a value with predefined unit of measurement.
The base class also has the following main properties:

```
  property PhysicalUnit: TUnit read fUnit write SetUnit;
```

```
  property StringValue: string read GetStringValue write
SetStringValue;
```

The first allows get or set new unit of measurement, the second allows get or set the string representation of the value.
There are the following implemented descendants of the base class: ***ScalarValue***, ***VectorValue***, ***TensorValue***. They have the ***Value*** property which holds real, 3D vector and 3D tensor values accordingly.

The main goal of introducing the physical value concept is working with dimensional values in program taking into account their 'physical dimension'. The behavior of the physical value is the following:

- When changing the unit of measurement its dimension cannot be changed.
- When changing the unit of measurement its value automatically converted into new unit.

Also, the ***PhysicalValue*** class allows easily manipulate with strings – convert string into a value and vice versa.

Physical value application

The main application of the physical value concept is taking into account the dimensions of all the numerical values in the program. The following code demonstrates an example of the physical value application: the water consumption measurement system, which can receive indications from different gauges with different units of measurement, and displays them always in liters per minute.

```
var
  s, err: string;
  v: TScalarValue;
  u: TUnit;
begin
  s:= '0.01 m^3/s'; // receive the gauge indication
  v:= TScalarValue.Create(s);
  u:= TScalarValue.UnitConverter.ConvertString('L/min', err);
  v.PhysicalUnit:= u;
  s:= v.StringValue;
  // display the gauge indication
end;
```

The received gauge indication 's' is parsed with the scalar value constructor. When the unit of measurement of the physical value changed, its numerical value automatically converted. The displayed value 's' will always show the indication measured in liters per minute, for this example value it is '600 L/min'.

Appendix A. Physics dimensions and quantities

Table A.1. Physical dimensions.

Dimension	Symbol	Virtuality[3]
Dimensionless (number)	1	-
Length	L	False
Time	T	False
Mass	M	False
Temperature	Θ	False
Electric Current	I	False
Amount of Substance	N	False
Luminous Intensity	J	False
Information	Y	True
Plane Angle	Φ	True
Solid Angle	Ω	True

Table A.2. List of physical quantities, defined in PHYSICS library.

Quantity	Symbol	Dimension
Length	L	L
Time	T	T
Mass	M	M
Temperature	Θ	Θ
Electric Current	I	I
Amount of Substance	N	N
Luminous Intensity	J	J
Information	Y	Y
Plane Angle	Φ	Φ
Solid Angle	Ω	Ω
Area	A	L^2
Volume	V	L^3
Frequency	F	1/T
Wavenumber	k	1/L
Wavelength	λ	L
Mean lifetime	τ	T
Speed	V	L/T
Angular Speed	ω	Φ/T
Acceleration	a	L/T^2

[3] Virtuality means that the dimension is equivalent to the dimensionless quantity in fact.

Quantity	Symbol	Dimension
Angular Acceleration	α	Φ/T^2
Density	ρ	M/L^3
Linear Density	ρ_1	M/L
Area Density	ρ_2	M/L^2
Specific volume	v	L^3/M
Weight	W	$L\ M/T^2$
Flow	F	L^3/T
Impulse	p	L M/T
Moment of inertia	I	L^2 M
Angular momentum	L	L^2 M/T
Spin	S	L^2 M/T
Viscosity	η	M/L T
Force	F	$L\ M/T^2$
Pressure	P	$M/L\ T^2$
Stress	σ	$M/L\ T^2$
Strain	ε	1
Moment	M	$L^2\ M/T^2$
Surface tension	Y	M/T^2
Energy	E	$L^2\ M/T^2$
Specific energy	e	L^2/T^2
Energy density	Eρ	$M/L\ T^2$
Work	W	$L^2\ M/T^2$
Power	W	$L^2\ M/T^3$
Heat	Q	$L^2\ M/T^2$
Heat flux density	Qφ	M/T^3
Thermal conductivity	k	$L\ M/T^3\ \Theta$
Molar heat capacity	cμ	$L^2\ M/T^2\ N$
Specific heat capacity	c	$L^2/T^2\ \Theta$
Entropy	S	$L^2\ M/T^2\ \Theta$
Enthalpy	H	$L^2\ M/T^2$
Chemical potential	μ	$L^2\ M/T^2\ N$
Reaction rate	r	$N/L^3\ T$
Molar concentration	C	N/L^3
Electric Charge	Q	T I
Electric Potential	P	$L^2\ M/T^3\ I$
Electric Resistance	R	$L^2\ M/T^3\ I^2$
Electrical conductance	G	$T^3\ I^2/L^2\ M$
Capacitance	C	$T^4\ I^2/L^2\ M$

Quantity	Symbol	Dimension
Electric Dipole moment	p	L T I
Current Density	J	I/L^2
Impedance	Z	L^2 M/T^3 I^2
Magnetic Flux	MF	L^2 M/T^2 I
Magnetic Field	MFi	M/T^2 I
Inductance	Ind	L^2 M/T^2 I^2
Magnetic flux density	B	M/T^2 I
Magnetization	M	I/L
Permeability	μ	M/L I^2
Permittivity	ε	T^4 I^2/L^2 M
Radioactivity	A	1/T
Dose equivalent	H	L^2/T^2
Radioactive Dose	D	L^2/T^2
Radiance	L	M/T^3 Ω
Radiant intensity	I	L^2 M/T^3 Ω
Irradiance	E	M/T^2
Illuminance	Ey	J $Ω/L^2$
Level (relative)	L	1
Entropy (information)	H	Y

Appendix B. Physics prefixes and units of measurement

Table B.1. Prefixes.

Prefix	Symbol	Value
Deci	d	1E-1
Centi	c	1E-2
Milli	m	1E-3
Micro	µ	1E-6
Nano	n	1E-9
Pico	p	1E-12
Femto	f	1E-15
Atto	a	1E-18
Zepto	z	1E-21
Yocto	y	1E-24
Deca	da	10
Hecto	h	100
Kilo	k	1000
Mega	M	1E+6
Giga	G	1E+9
Tera	T	1E+12
Peta	P	1E+15
Exa	E	1E+18
Zetta	Z	1E+21
Yotta	Y	1E+24
Kibi	Ki	1024
Mebi	Mi	1024^2
Gibi	Gi	1024^3
Tebi	Ti	1024^4
Pebi	Pi	1024^5

Table B.2. List of units of measurement, defined in PHYSICS library.

Unit	Symbol	Dimension
Percent	%	1
Permille	‰	1
Metre	m	L
Atomic unit of length	a_0	L
Angstrom	Å	L
Micron	µ	L
Light Year	ly	L

Unit	Symbol	Dimension
Mile (international)	mi	L
Foot	ft	L
Inch	in	L
Yard	yd	L
Point	pt	L
Second	s	T
Atomic unit of time	au	T
Minute	min	T
Hour	h	T
Kilogram	kg	M
Gram	g	M
Atomic mass unit	AMU	M
Electron rest mass	m_0	M
Electronvolt	eV	M
Carat	kt	M
Carat (metric)	ct	M
Tonne	t	M
Kelvin	ºK	Θ
Celsius	ºC	Θ
Fahrenheit	ºF	Θ
Delisle	ºDe	Θ
Newton (temperature)	ºN	Θ
Rankine	ºR	Θ
Reaumur	ºRe	Θ
Romer	ºRo	Θ
Ampere	A	I
Mole	mol	N
Candela	cd	J
Bit	b	Y
Byte	B	Y
Nibble	nib	Y
Trit	trit	Y
Dit	dit	Y
Nat	nat	Y
Radian	rad	Φ
Degree	º	Φ
Arcminute	'	Φ
Arcsecond	"	Φ

Unit	Symbol	Dimension
Steradian	sr	Ω
Are	a	L^2
Hectare	ha	L^2
Litre	L	L^3
Hertz	Hz	1/T
Revolutions per minute	rpm	1/T
Light speed	c	L/T
Standard gravity	g_0	L/T^2
Newton	N	$L\ M/T^2$
Dyne	dyn	$L\ M/T^2$
Kilogram-force	kgf	$L\ M/T^2$
Sthene	sn	$L\ M/T^2$
Ounce-force	ozf	$L\ M/T^2$
Pound-force	lbf	$L\ M/T^2$
Poundal	pdl	$L\ M/T^2$
Ton-force	tnf	$L\ M/T^2$
Pascal	Pa	$M/L\ T^2$
Atmosphere	atm	$M/L\ T^2$
Bar	bar	$M/L\ T^2$
Millimetre of mercury	mmHg	$M/L\ T^2$
Millimetre of water	mmH_2O	$M/L\ T^2$
Pieze	pz	$M/L\ T^2$
Pound per square foot	psf	$M/L\ T^2$
Pound per square inch	psi	$M/L\ T^2$
Torr	torr	$M/L\ T^2$
Joule	J	$L^2\ M/T^2$
Erg	erg	$L^2\ M/T^2$
Watt	W	$L^2\ M/T^3$
Stokes	St	L^2/T
Litre per minute	LPM	L^3/T
Coulomb	C	T I
Atomic unit of charge	e	T I
Volt	V	$L^2\ M/T^3\ I$
Abvolt	abV	$L^2\ M/T^3\ I$
Statvolt	statV	$L^2\ M/T^3\ I$
Ohm	O	$L^2\ M/T^3\ I^2$
Farad	F	$T^4\ I^2/L^2\ M$
Weber	Wb	$L^2\ M/T^2\ I$

Unit	Symbol	Dimension
Tesla	T	$M/T^2\ I$
Henry	H	$L^2\ M/T^2\ I^2$
Gauss	G	$M/T^2\ I$
Maxwell	Mx	$L^2\ M/T^2\ I$
Debye	D	L T I
Lumen	lm	J Ω
Lux	lx	$J\ Ω/L^2$
Phot	ph	$J\ Ω/L^2$
Footcandle	fc	$J\ Ω/L^2$
Sievert	Sv	L^2/T^2
Rontgen equivalent man	rem	L^2/T^2
Gray	Gy	$L^2\ M^2/T^2$
Rad	Rad	$L^2\ M^2/T^2$
Roentgen	R	T I/M
Becquerel	Bq	1/T
Curie	Ci	1/T
Rutherford	rd	1/T
Decibel (10)	dB_{10}	1
Decibel (20)	dB_{20}	1
Bell	bel	1
Neper	Np	1
Binary logarithm	$\log_2$	1
Decimal logarithm	$\log_{10}$	1
Natural logarithm	$\log_e$	1
Decibel-milliwatts	dBm	$L^2\ M/T^3$
Decibel-Joules	dBJ	$L^2\ M/T^2$
Decibel-Pascals	dBPa	$M/L\ T^2$
Decibel SPL (air)	dBSPL	$M/L\ T^2$
Decibel SPL (liquid)	dBSPLl	$M/L\ T^2$
Decibel-Volts	dBV	$L^2\ M/T^3\ I$
Decibel-Volts (unloaded)	dBu	$L^2\ M/T^3\ I$