ZSQtLib

Physikalische Werte und Einheiten

Benutzerhandbuch

Copyright

Marken

Inhaltsverzeichnis

1. Grundlagen

Eine physikalische Größe ist eine quantitativ bestimmbare Eigenschaft eines physikalischen Objektes. Sie ist entweder direkt messbar (*Messgröße*) oder kann aus anderen Messgrößen berechnet werden (*abgeleitete Größe*). Den Zusammenhang zwischen physikalischen Größen vermitteln physikalische Gesetze.

Unterscheidungsmerkmal zwischen gleichartigen physikalischen Größen ist ihr Größenwert oder Messwert, der als Produkt aus Zahlenwert (auch Maßzahl genannt) und Maßeinheit angegeben wird. Unabhängige Größen bilden zusammen mit allen aus ihnen ableitbaren Größen ein Größensystem.

Größenart

Die Definition einer Größe erfordert die Angabe einer reproduzierbaren Messvorschrift. Wenn die gleiche (oder eine äquivalente) Messvorschrift zur Definition verschiedener Größen genutzt werden kann, so bezeichnet man die Größen als *gleichartig*. Ausschließlich innerhalb der gleichen Größenart lassen sich Größen sinnvoll quantitativ vergleichen, addieren und subtrahieren.

Beispielsweise sind Breite, Höhe und Länge eines Quaders, Durchmesser eines Rohrs, Wellenlänge usw. Größen der Größenart „Länge“, da sie alle mit der gleichen Messvorschrift gemessen werden können.

Größenwert

Das Unterscheidungsmerkmal zwischen Größen der gleichen Größenart ist ihr *Größenwert*. Dieser beschreibt eine bestimmte Eigenschaft eines Objektes quantitativ und erlaubt somit die Vergleichbarkeit von Objekten gleicher Eigenschaft.

Zahlenwert und Einheit

Die Bestimmung des Größenwerts erfolgt technisch über den Vorgang einer Messung. Hierbei wird das Verhältnis des Größenwerts zu dem Wert einer gleichartigen, feststehenden und wohl definierten Vergleichsgröße ermittelt. Den Vergleichsgrößenwert bezeichnet man als *Maßeinheit* oder kurz *Einheit*, den Quotienten aus den Werten der zu quantifizierenden Größe und der Vergleichsgröße als *Zahlenwert* oder *Maßzahl*. Der Größenwert einer Größe kann dann als Produkt aus Zahlenwert und Einheit dargestellt werden.

Die Bestimmung des Größenwerts erfolgt entweder durch eine direkte Messung oder aus der Berechnung aus anderen Messgrößen. Die Definition einer Einheit unterliegt der menschlichen Willkür. Theoretisch ist es ausreichend, eine einzige Einheit für eine Größenart zu definieren. Historisch bedingt haben sich aber häufig eine Vielzahl verschiedener Einheiten für die gleiche Größenart gebildet. Diese unterscheiden sich lediglich um einen reinen Zahlenfaktor.

Formel- und Einheitenzeichen

Einer physikalischen Größe wird in mathematischen Gleichungen ein Schriftzeichen zugeordnet, das man *Formelzeichen* nennt. Dieses ist grundsätzlich willkürlich, jedoch existieren eine Reihe von Konventionen (z.B. DIN 1304) zur Bezeichnung bestimmter Größen. Üblicherweise besteht ein Formelzeichen nur aus einem einzigen Buchstaben, der zur weiteren Unterscheidung mit einem Index versehen werden kann.

Auch für Einheiten gibt es standardisierte Schriftzeichen, die Einheitenzeichen genannt werden. Sie bestehen meistens aus einem oder mehreren lateinischen Buchstaben oder seltener aus einem Sonderzeichen wie z.B. einem Gradzeichen.

Die Angabe des Größenwerts erfolgt immer als Produkt aus Zahlenwert und Einheit.

Da der Zahlenwert von der gewählten Maßeinheit abhängt, ist die alleinige Darstellung des Formelzeichens nicht eindeutig. Deshalb ist für die Beschriftung von Tabellen und Koordinatenachsen die Darstellung „G/[G]“ (z.B. „*m*/kg“) oder „G in [G]“ (z.B. „*m* in kg“) üblich.

Formatierung

Die Formatierung ist durch DIN 1338 geregelt. Demnach wird das Formelzeichen *kursiv* geschrieben, während das Einheitenzeichen mit aufrechter Schrift geschrieben wird, um es vom Formelzeichen zu unterscheiden. Beispielsweise bezeichnet „*m*“ das Formelzeichen für die Größe „Masse“ und „m“ das Einheitenzeichen für die Maßeinheit „Meter“.

Zwischen der Maßzahl und dem Einheitenzeichen wird ein Leerzeichen geschrieben. Eine Ausnahme von dieser Regel stellen die Gradzeichen dar, die ohne Zwischenraum direkt hinter die Maßzahl geschrieben werden („ein Winkel von 180°“), sofern keine weiteren Einheitenzeichen folgen („die Außentemperatur beträgt 23 °C“).

Fehlerbehaftete Größen

Bei fehlerbehafteten Größenwerten wird der Zahlenwert mit seiner Messunsicherheit angegeben, meistens in Form des mittleren Fehlers oder manchmal – falls bekannt – des Maximalfehlers. Das Kenntlichmachen geschieht meistens durch ein „±“ nach dem fehlerbehafteten Zahlenwert, gefolgt von dem Fehlerwert (wobei Klammern erforderlich sind, sofern eine Einheit folgt, damit diese sich auf beide Werte bezieht).

„P = (12,34 ± 0,23) mW“

Die Anzahl der anzugebenden unsicheren Dezimalstellen des Zahlenwerts richtet sich nach dem Fehlerwert. Beginnt dieser mit einer 1 oder 2, so werden zwei Stellen notiert, ansonsten nur eine. Gegebenenfalls ist der Zahlenwert zu runden.

Nicht unüblich ist auch die Angabe der Messunsicherheit als Verhältniszahl, z.B. in Prozent.

Auflösung

Handelt es sich bei Größenwerten nicht um Messgrößen, sondern um Einstellparameter, so können diese nicht beliebig genau vorgenommen werden. In diesem Fall spricht man eher von einer Einstellgenauigkeit bzw. Auflösung des Größenwerts. Hat man den Parameter gemäß seiner Auflösung eingestellt, so geschieht dies zusätzlich nur mit einer gewissen Genauigkeit. Für Einstellparameter müsste also sowohl deren Auflösung als auch deren Genauigkeit berücksichtigt werden. Die Auflösung eines Einstellparameters ist aber immer „gröber“ als die Genauigkeit. Es macht ja keinen Sinn, einen Einstellwert mit einer Auflösung von „0.001 Volt“ anzubieten, wenn der Wert dann tatsächlich im Gerät nur mit einer Genauigkeit von „0.1 Volt“ eingestellt wird. Im Grunde liegt also sowohl bei der Auflösung als auch der Messunsicherhet derselbe Sachverhalt vor. Im Falle von Einstellparameter sind die Werte ebenfalls mit einem Fehler – einer Einstellunsicherheit - behaftet.

Bei der Anzahl der anzugebenden Dezimalstellen muss aber zwischen Mess- und Einstellwerten unterschieden werden. Für Einstellparameter ist es nicht sinnvoll, mehr Stellen wiederzugeben, als durch die Auflösung des Wertes vorgegeben ist. Ferner können Einstellparameter von Benutzern eines Messgerätes über ein Eingabefeld inkrementell in Schritten verändert werden. Die Schrittweite muss sich dabei nach der Auflösung der Einstellwerte richten. Kann ein Parameter z.B. nur mit einer Genauigkeit von 0,25 Volt eingestellt werden, ist ein Einstellwert von 2,1 Volt nicht möglich. Stattdessen sind nur Werte möglich, die ganzzahlig durch 0,25 teilbar sind (z.B. 7,0 -> 7,25 -> 15,75 etc.).

Generell verfügen Einstellparameter über einen maximalen und einen minimalen Wert. Die Schrittweite gemäß der Auflösung beginnt somit beim minimal möglichen Wert. Kann z.B. eine Spannung nur im Bereich -4 V bis +5 V mit einer Auflösung von 1,5 V eingestellt werden, sind lediglich die Werte -4,0 -2,5 -1,0 0,5 2,0 3,5 und 5,0 möglich.

Verknüpfung zwischen physikalischen Größen

Größengleichungen

Die Darstellung von Naturgesetzen und technischen Zusammenhängen in mathematischen Gleichungen nennt man Größengleichungen. Die Formelzeichen einer Größengleichung haben die Bedeutung physikalischer Größen, sofern sie nicht als Symbole für mathematische Funktionen oder Operatoren gemeint sind. Größengleichungen gelten unabhängig von der Wahl der Einheiten.

Größengleichungen verknüpfen verschiedene physikalische Größen und deren Größenwerte. Zur Auswertung muss man die Formelzeichen durch das Produkt aus Zahlenwert und Einheit ersetzen. Die verwendeten Einheiten sind dabei unerheblich. Die Größenart muss auf beiden Seiten des Gleichheitszeichens jedoch übereinstimmen, damit die Gleichung physikalisch sinnvoll ist.

Beispiele:

F = ma.....Kraft = Masse * Beschleunigung

P = UI .....Leistung = Spannung * Stromstärke
R = U/I.....Widerstand = Spannung / Stromstärke
U = RI.....Spannung = Widerstand * Stromstärke
P = I²R .....Leistung = (Stromstärke im Quadrat) * Widerstand
R = P/I² .....Widerstand = Leistung / (Stromstärke im Quadrat)
usw.

Rechenregeln

Für physikalische Größen sind nicht alle Rechenoperationen sinnvoll, die auch mit reinen Zahlen möglich wären. Es hat sich erwiesen, dass eine geringe Anzahl Rechenregeln ausreicht, um alle bekannten Naturgeschehen zu beschreiben.

Addition und Subtraktion ist nur zwischen Größen der gleichen Größenart möglich.

Multiplikation und Division sowohl von verschiedenen Größen als auch mit reinen Zahlen sind uneingeschränkt möglich. Häufig ist das Produkt bzw. der Quotient eine neue physikalische Größe. Damit sind auch Potenzen mit ganzzahligen Exponenten erlaubt. Das Ziehen der Quadratwurzel aus einer Größe ist nur dann möglich, wenn die Größe sich als Produkt zweier gleichartiger Größen darstellen lässt.

Transzendente Funktionen wie „exp“, „log“, „sin“, „tanh“, usw. sind nur für reine Zahlen definiert und damit nur bei dimensionslosen Größen möglich. Unter diese dimensionslosen Größen fallen die logarithmischen Verhältnisgrößen wie Neper, Bel und Dezibel, wie sie z.B. bei Angaben für Lautstärken oder in der Nachrichtentechnik bei der Angabe für Leistungen verwendet werden.

1.1. Größen- und Einheitensysteme

Größensysteme

Jedes Wissensgebiet der Technik und Naturwissenschaften wird durch einen beschränkten Satz an physikalischen Größen beschrieben, die über Naturgesetze miteinander verknüpft sind. Die zugrunde liegenden Größen bilden ein *Größensystem*. Man teilt die Größen dieses Systems in *Basisgrößen* und *abgeleitete Größen*. Der Unterschied liegt darin, dass sich die abgeleiteten Größen als Potenzprodukte der Basisgrößen darstellen lassen, während das bei den Basisgrößen nicht möglich ist. Diese Einteilung ist weitgehend willkürlich und geschieht meistens aus praktischen Gründen. Die Anzahl der Basisgrößen bestimmt den *Grad* des Größensystems. Beispielsweise ist das internationale Größensystem mit seinen sieben Basisgrößen ein *Größensystem siebten Grades*

Einheitensysteme

Man benötigt für jede Größe eine Einheit, um den Größenwert angeben zu können. Daher entspricht jedem Größensystem ein *Einheitensystem* gleichen Grades, das sich analog aus untrennbaren *Basiseinheiten* und weiter aufteilbare *abgeleitete Einheiten* zusammensetzt. Die abgeleiteten Einheiten werden aus den Basiseinheiten durch Produkte von Potenzen dargestellt, im Unterschied zu Größensystemen jedoch eventuell ergänzt durch einen

Zahlenfaktor. Man bezeichnet das Einheitensystem als *kohärent* (zusammenhängend), wenn alle Einheiten ohne diesen zusätzlichen Faktor gebildet werden können. In derartigen Systemen können alle Größengleichungen als Zahlenwertgleichungen aufgefasst und dementsprechend schnell ausgewertet werden.

Das in fast allen Ländern der Welt benutzte internationale Einheitensystem (SI) ist ein kohärentes Einheitensystem siebten Grades, das auf dem internationalen Größensystem fußt. Das SI definiert zudem standardisierte Vorsätze für Maßeinheiten. Allerdings sind die so gebildeten Vielfachen oder Teile einer SI-Einheit selbst nicht Teil des eigentlichen Einheitensystems, da dies der Kohärenz widerspräche. Beispielsweise ist ein fiktives Einheitensystem, das die Basiseinheiten Zentimeter (cm) und Sekunde (s) sowie die abgeleitete Einheit Meter pro Sekunde (m/s) umfasst, nicht kohärent: Wegen

$$1\ \mathrm{m/s} = 100\ \mathrm{cm/s}$$

benötigt man einen Zahlenfaktor (100) bei der Bildung dieses Systems.

1.2. Besondere Größen

Quotienten- und Verhältnisgrößen

Der Quotient zweier Größen ist eine neue Größe. Eine solche Größe bezeichnet man als *Verhältnisgröße*, wenn die Ausgangsgrößen von der gleichen Größenart sind, ansonsten als *Quotientengröße*.

Verhältnisgrößen sind grundsätzlich *dimensionslos*. Der Name einer Verhältnisgröße beinhaltet meistens ein Adjektiv wie *relativ* oder *normiert* oder er endet auf *-zahl* oder *-wert*. Beispiele sind die Reynoldszahl und der CW-Wert.

Verschiedene Verhältnisgrößen gehören nur in seltenen Fällen zur gleichen Größenart. Manchmal werden daher zur besseren Trennung bei der Angabe ihres Größenwerts die Einheitenzeichen nicht gekürzt. Häufig werden Verhältnisgrößen in den Einheiten %, ‰ oder *ppm* angegeben. Eine besondere Stellung haben Verhältniseinheiten, wenn sie das Verhältnis gleicher Einheiten sind. Diese sind immer 1 und damit *idempotent*, d.h., sie können beliebig oft mit sich selbst multipliziert werden, ohne ihren Wert zu ändern. Einige idempotente Verhältniseinheiten tragen besondere Namen, wie beispielsweise die Winkeleinheit *Radiant (rad)*. In kohärenten Einheitensystemen sind die Verhältniseinheiten immer 1, also idempotent.

Idempotente Verhältniseinheiten sind deshalb interessant, weil man hier die Zahlenwerte einfach multiplizieren kann. Sagt man, dass ein Anteil von 0,3 der Erdoberfläche Landmassen sind und der Kontinent Asien einen Anteil von 0,3 der Landmasse darstellt, kann man folgern, dass 0,09 der Erdoberfläche vom Kontinent Asien bedeckt sind, weil wir hier die Einheit 1 haben, die idempotent ist. Im Gegensatz dazu sind Angaben in Prozent nicht idempotent. Sagt man beispielsweise, dass 30 % der Erdoberfläche Landmassen sind und der Kontinent Asien 30 % der Landmasse darstellt, kann man nicht folgern, dass 900 % der Erdoberfläche vom Kontinent Asien bedeckt sind, weil % nicht idempotent ist, also $\%^2$ nicht dasselbe wie % ist.

In vielen technischen Bereichen sind die *logarithmierten Verhältnisse* von besonderem Interesse. Derartige Größen werden als *Pegel* oder *Maß* bezeichnet. Wird bei der Bildung der natürliche Logarithmus verwendet, so kennzeichnet man dieses durch die Hilfseinheit *Neper* (Np), ist es der dekadische Logarithmus, so nutzt man die Hilfseinheit *Bel* (B) bzw. häufiger ihr Zehntel, das *Dezibel* (dB).

1.3. Logarithmische Einheiten

Das **Bel** (B) ist eine nach Alexander Graham Bell benannte Hilfsmaßeinheit zur Kennzeichnung von Pegeln und Maßen. Diese logarithmischen Größen finden ihre Anwendung unter anderem in der Akustik (z. B. Schalldruckpegel, Schalldämm-Maß), der Hochfrequenztechnik als Teil der Nachrichtentechnik (z. B. SNR), der Tontechnik und der Automatisierungstechnik. In der Praxis ist die Verwendung des zehnten Teils eines Bels (**Dezibel**, Einheitenzeichen **dB**) üblich.

Das Bel dient zur Kennzeichnung des dekadischen Logarithmus des Verhältnisses zweier gleichartiger Leistungs- bzw. Energiegrößen P_1 und P_2:

$$L = \left(\lg \frac{P_2}{P_1}\right) \mathrm{B} = 10 \left(\lg \frac{P_2}{P_1}\right) \mathrm{dB}$$

Für L ergibt sich z. B. der Wert ein Bel (B), wenn das Leistungsverhältnis $P_2 / P_1 = 10$ ist. Das gebräuchlichere Dezibel (dB) wird mit Hilfe des Einheitenvorsatzes „Dezi“ (Symbol „d“) gebildet:

$$1\,\mathrm{dB} = \frac{1}{10}\,\mathrm{B}$$

In linearen Systemen verhalten sich die Leistungs- bzw. Energiegrößen P proportional zu den Quadraten der einwirkenden Effektivwerte von Feldgrößen x (z. B. elektrische Spannung, Schalldruck), d. h.

$$P \sim \tilde{x}^2$$

Soll von Feldgrößen ausgehend ein Pegel oder Maß berechnet werden, steht dadurch das Verhältnis der Quadrate dieser Größen und es gilt

$$L = 10 \lg \frac{P_2}{P_1}\,\mathrm{dB} = 10 \lg \frac{\tilde{x}_2^2}{\tilde{x}_1^2}\,\mathrm{dB} = 20 \lg \frac{\tilde{x}_2}{\tilde{x}_1}\,\mathrm{dB}$$

Zu beachten ist dabei, dass das Argument der lg-Funktion eine dimensionslose Größe sein muss, d. h. die Größen P_1 und P_2 bzw. x_1 und x_2 stets die gleiche Einheit haben müssen. Ein Beispiel für eine so definierte Größe ist der Schalldruckpegel.

Umrechnung in die Einheit Neper

Dezibel und Neper stehen in einem festen linearen Verhältnis zueinander:

$$L = 20 \lg \frac{\tilde{x}_2}{\tilde{x}_1}\,\mathrm{dB} = \ln \frac{\tilde{x}_2}{\tilde{x}_1}\,\mathrm{Np}$$

Durch Umstellen nach dB und Umrechnung der Logarithmenbasis folgt daraus:

$$1\,\mathrm{dB} = \frac{\ln \frac{\tilde{x}_2}{\tilde{x}_1}\,\mathrm{Np}}{\frac{20}{\ln 10} \ln \frac{\tilde{x}_2}{\tilde{x}_1}} = \frac{\ln 10}{20}\,\mathrm{Np} \approx \frac{1}{8{,}686}\,\mathrm{Np} \approx 0{,}115\,\mathrm{Np}$$

Dezibel und Neper, Historische Enwicklung

Obwohl nicht das Bel bzw. Dezibel, sondern das Neper die zum Internationalen Einheitensystem SI kohärente Hilfsmaßeinheit für logarithmische Verhältnisgrößen ist, wird in der Praxis überwiegend das Dezibel verwendet. Das hat zum einen historische Gründe: In den USA war bis 1923 als Einheit für das Dämpfungsmaß einer Fernsprechverbindung die Hilfsmaßeinheit „Mile Standard Cable“ (m.s.c.) in Verwendung. Diese Einheit entspricht dem Dämpfungsmaß eines bestimmten Kabeltyps („19 gauge“) bei einer Länge von einer englischen Meile und einer Frequenz von 800 Hz und gleichzeitig der mittleren subjektiven Wahrnehmbarkeitsschwelle beim Vergleich von zwei Lautstärken. Letzteres trifft ebenfalls für das Dezibel zu. Deshalb ergaben sich bei Verwendung des Dezibels in etwa die gleichen Zahlenwerte wie bei Verwendung von „Mile Standard Cable“ (1 m.s.c. = 0,9221 dB). Ein weiterer Grund für die bevorzugte Verwendung des Dezibels ist, dass sich einfacher fassbare Zahlenwerte ergeben, so entspricht z. B. die Verdoppelung der Leistung einer Änderung von etwa 3 dB, die Verzehnfachung einer Änderung von 10 dB.

Verwendung mit anderen Maßeinheiten, Anhängsel

So wie jede andere Maßeinheit kann das Bel bzw. Dezibel zusammen mit anderen Maßeinheiten verwendet werden, wenn damit eine Größe beschrieben wird, bei der ein Pegel oder Maß durch Multiplikation oder Division mit einer anderen Größe verknüpft wird. Beispiele dafür sind das Dämpfungsmaß einer Leitung in Dezibel pro Meter (dB/m) oder der bezogene Schallleistungspegel in Dezibel pro Quadratmeter (dB/m^2).

Obwohl es nach den für Größen geltenden Rechenregeln nicht korrekt ist, Anhängsel an eine Einheit anzubringen, um Informationen über die Art der betrachteten Größe mitzuteilen, sind solche Anhängsel beim Dezibel gebräuchlich. Wegen der Eindeutigkeit und der möglichen Verwechslungsgefahr mit Einheitenprodukten (z. B. dB•m statt dBm) sind diese Informationen stets mit der Größe und nicht mit der Einheit zu verknüpfen. Die geläufigsten Beispiele für dB-Anhängsel sind in der folgenden Tabelle zusammengefasst:

Einheit mit Anhängsel	Bedeutung	Empfohlene Schreibweisen (DIN, IEC, ISO)	
		Hinweis an der Größe	Hinweis am Einheitenzeichen
dBu	Spannungspegel mit der Bezugsgröße $\sqrt{600\,\Omega \cdot 0{,}001\,\mathrm{W}} \approx 0{,}7746\,\mathrm{V}$	$L_u(\mathrm{re}\ 0{,}775\,\mathrm{V}) = \ldots\,\mathrm{dB}$	$L_u = \ldots\,\mathrm{dB}\,(0{,}775\,\mathrm{V})$
dBV	Spannungspegel mit der Bezugsgröße	$L_V(\mathrm{re}\ 1\,\mathrm{V}) = \ldots\,\mathrm{dB}$	$L_V = \ldots\,\mathrm{dB}\,(\mathrm{V})$
dBA	A-bewerteter Schalldruckpegel	$L_{pA}(\mathrm{re}\ 20\,\mu\mathrm{Pa}) = \ldots\,\mathrm{dB}$	$L_p = \ldots\,\mathrm{dB}\,(\mathrm{A})$
	A-bewerteter Schallleistungspegel	$L_{WA}(\mathrm{re}\ 1\,\mathrm{pW}) = \ldots\,\mathrm{dB}$	$L_W = \ldots\,\mathrm{dB}\,(\mathrm{A})$
dBm	Leistungspegel mit der Bezugsgröße 1 mW	$L_P(\mathrm{re}\ 1\,\mathrm{mW}) = \ldots\,\mathrm{dB}$	$L_P = \ldots\,\mathrm{dB}\,(1\,\mathrm{mW})$
dBW	Leistungspegel mit der Bezugsgröße 1 W	$L_P(\mathrm{re}\ 1\,\mathrm{W}) = \ldots\,\mathrm{dB}$	$L_P = \ldots\,\mathrm{dB}\,(1\,\mathrm{W})$
dBµ	Pegel der elektrischen Feldstärke mit der Bezugsgröße 1 µV/m	$L_E(\mathrm{re}\ 1\,\mu\mathrm{V/m}) = \ldots\,\mathrm{dB}$	$L_E = \ldots\,\mathrm{dB}\,(1\,\mu\mathrm{V/m})$

2. Das Klassenmodell

Das Subsystems *ZSPhysVal* der *ZSQtLib* ist eine C++ Klassenbibliothek, die den Umgang mit physikalischen Werten erleichtern soll und dabei die in der Einführung beschriebenen theoretischen Grundlagen berücksichtigt.

2.1. Organisation der Einheiten in einem Objekt-Baum

Das Datenmodel physikalischer Einheiten soll die Unterteilung der Einheiten in die SI-Basisgrößen, in Wissensgebiete, Einheitensysteme, den zugehörigen SI-Basiseinheiten und den daraus abgeleiteten Einheiten widerspiegeln.

Die SI-Basisgrößen und Basiseinheiten sind:

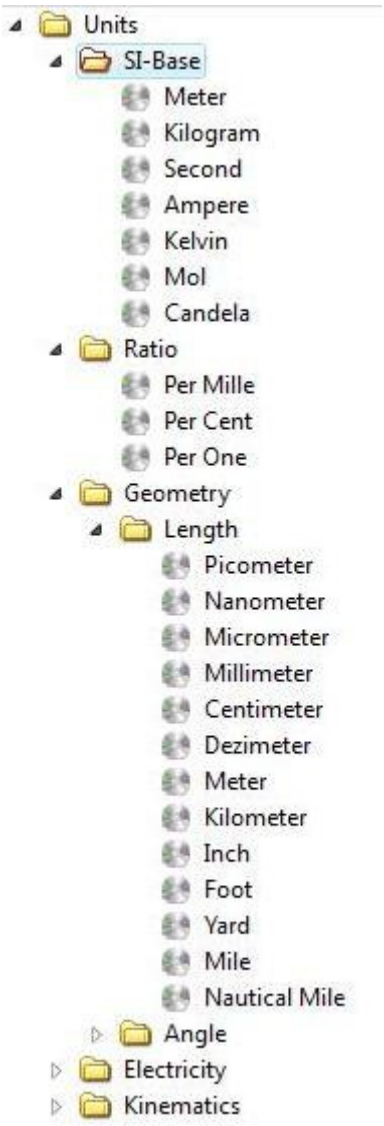
Basisgröße	**Formelzeichen**	**Symbol für Dimension**	**Basiseinheiten**	**Einheitensymbol**
Länge	***l, s, x, r, etc.***	***L***	Meter	***m***
Masse	***m***	***M***	Kilogramm	***kg***
Zeit	***t***	***T***	Sekunde	***s***
elektrische Stromstärke	***I***	***I***	Ampere	***A***
Absolute Temperatur	***T***	***θ***	Kelvin	***K***
Stoffmenge	***n***	***N***	Mol	***mol***
Lichtstärke	I_V	***J***	Candela	***cd***

Physikalische Einheiten lassen sich in acht Wissensgebiete unterteilen, für die die nachfolgende Tabelle eine (unvollständige) Übersicht gibt:

Wissensgebiet	Größenart	Physikalische Größe	Formelzeichen	Dimension	SI-Einheit
Geometrie	Winkel	Ebener Winkel, Drehwinkel	$\alpha, \beta, \gamma, \ldots$ $\varphi, \theta, ..$	**L**	Radiant (rad)
	Länge	Länge	l	**L**	Meter (m)
	Länge	Durchmesser	d, D	**L**	Meter (m)
	…	…	…	…	…
Kinematik	Zeit	Zeit, Zeitspanne, Dauer	t	**T**	Sekunde (s)
	Zeit	Periodendauer	T, τ	**T**	Sekunde (s)
	Geschwindigkeit	Geschwindigkeit	v, u, w, c	**L/T**	m/s
	Beschleunigung	Beschleunigung	a	$\mathbf{L/T^2}$	m/s^2
	Frequenz	Frequenz	f, ν	**L/T**	Hertz (Hz)
	…	…	…	…	…
Mechanik	Masse	Masse	m	**M**	Kilogramm (kg)
	Kraft	Kraft	F	$\mathbf{ML/T^2}$	Newton (N)
	Kraft	Reibung	F_R	$\mathbf{ML/T^2}$	Newton (N)
	Leistung	Leistung	P	$\mathbf{ML^2/T^3}$	Watt (W)
	…	…	…	…	…
Thermodynamik	Temperatur	Absolute Temperatur	T	θ	Kelvin (K)
	Energie	Wärme, Wärmemenge	Q	$\mathbf{ML/T^2}$	Joule (J)
	Energie	thermische Energie	E_{th}	$\mathbf{ML/T^2}$	Joule (J)
	Leistung	Wärmestrom	Φ_{th}, Φ, Q	$\mathbf{ML^2/T^3}$	Watt (W)
	…	…	…	…	…
Elektrizität und Magnetismus	El. Stromstärke	El. Stromstärke	I	**I**	Ampere (A)
	El. Spannung	El. Spannung	U	$\mathbf{ML^2/T^3I^1}$	Volt (V)
	El. Widerstand	Ohmscher Widerstand	R	$\mathbf{ML^2/T^3I^2}$	Ohm (Ω)
	Leistung	Wirkleistung	P	$\mathbf{ML^2/T^3}$	Watt (W)
	…	…	…	…	…
Atomar und Molekular	Stoffmenge	Stoffmenge	n	**N**	Mol (mol)
	Relative Masse	Relative Atommasse	A_r	**1**	Eins
	…	…	…	…	…
Kernphysik	Aktivität	Aktivität	A	**1/T**	Becquerel (Bq)
	Zeit	Halbwertszeit	$T_{½}$	**T**	Sekunde (s)
	…	…	…	…	…
Fotometrie und Optik	Lichtstärke	Lichtstärke	I_V	**J**	Candela (cd)
	Länge	Brennweite	f	**L**	Meter (m)
	…	…	…	…	…

Ferner müssen noch die Quotienten- und Verhältnisgrößen wie Prozentangaben berücksichtigt werden.

Zur Verwaltung der Einheiten eignet sich eine Baumstruktur. Die Äste in dieser Baumstruktur bilden die Wissensgebiete und die Einheitensysteme wieder, die Blätter entsprechen den SI-Basisgrößen, den SI-Basiseinheiten und den daraus abgeleiteten Einheiten. Um die Einheiten in einer Baumstruktur zu organisieren bietet sich die Klasse *CModelObjPool* der ZS Systembibliothek an.



Mit den bisherigen Informationen ließe sich bereits ein erstes Klassendiagramm erstellen. Um die Einheiten über das Object Pool Model verwalten zu können, muss die Einheitenklasse von *QObject* abgeleitet werden. Da *QObject* nur 8 Byte Speicher benötigt, ist das durchaus akzeptabel. Sowohl die SI-Basiseinheiten als auch Verhältnisgrößen und alle abgeleiteten Einheiten innerhalb der Einheitensysteme haben gemeinsame Attribute, wie z.B. Symbol oder Namen. Deshalb wird eine Basisklasse *CUnit* definiert, deren Eigenschaften sowohl die Klasse für die SI-Basiseinheiten, die Klasse für die Verhältnisgrößen und die Klasse für die abgeleiten Einheiten erben.

Abgeleitete Einheiten innerhalb einer Größenart werden innerhalb einer Klasse *CPhysSize* organisiert, die – wie allerdings erst noch später genauer erläutert wird – noch weitere Aufgaben als die bloße Organisation der Einheiten einer Größenart übernehmen wird. Es

bietet sich an, für jede Größenart – wie z.B. der elektrischen Leistung - eine von *CPhysSize* abgeleitete Klasse zu erstellen. Diese abgeleiteten *CPhysSize*-Klassen sind aber nicht mehr Bestandteil der *ZSPhysVal* Klassenbibliothek, denn in diesem Falle müssten ja Klassen für alle Größenarten implementiert und bereitgestellt werden. Darüber hinaus müssten innerhalb einer Größenart alle abgeleiteten Einheiten implementiert werden – aber wer braucht schon gleichzeitig „Yokto-Meter“ ($=1*10^{-24}$ m) und „Yotto-Meter“ ($=1*10^{24}$ m) sowie „Yokto-Ampere“ ($=1*10^{-24}$ A) und „Yotto-Ampere“ ($=1*10^{24}$ A) in ein und derselben Applikation?

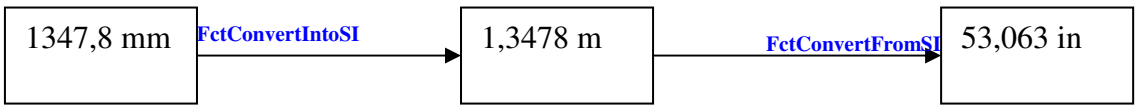
In der Regel wird nur eine kleine Teilmenge der Einheiten und Größenarten innerhalb einer Applikation verwendet werden. Deshalb ist es wohl eher sinnvoller, z.B. für jedes Wissensgebiet eine eigene Dll zu erstellen, in der nur die abgeleiteten Einheiten definiert sind, die in der Applikation auch benötigt werden.

2.2. Umrechnungen zwischen Einheiten derselben Größenart

Bislang wurde ein statisches Modell entworfen, mit dem sich die Einheiten in einer hierarchisch gegliederten Baumstruktur organisieren lassen. Mit Hilfe des Model/View Ansatzes von Qt 4 ist es damit möglich, mit relativ geringem Aufwand eine Benutzeroberfläche zu realisieren, mit der man durch die Einheiten-Konfiguration „browsen“ kann, um so die Konfiguration der Einheiten zu prüfen.

Doch die eigentliche Funktionalität fehlt noch. Der Anwendungsfall, einen Größenwert der Einheit A in den entsprechenden Größenwert der Einheit B zu konvertieren, wurde bisher noch nicht berücksichtigt.

Um eine Einheit in eine andere Einheit derselben Größenart umzurechnen, würde es ausreichen, für jede von der SI-Einheit abgeleitete Einheit zwei Umrechnungsfunktionen bereitzustellen: eine Funktion, mit der die abgeleitete Einheit in die SI-Einheit der Größenart umgerechnet wird und eine Funktion, mit der die SI-Einheit in die abgeleitete Einheit umgerechnet wird. Um auf diese Weise z.B. Millimeter in Inch umzurechnen, würde über die *FctConvertIntoSI* Umrechnungsfunktion der Einheit Millimeter der Wert zunächst in Meter umgerechnet und anschließend würde über die *FctConvertFromSI* Umrechnungsfunktion der Einheit Inch der Wert in Meter in die Einheit Inch umgerechnet.



Damit wären allerdings immer zwei Rechenschritte notwendig. Bei Größenarten mit nur linearen Einheiten, in denen die Einheiten mit reinen Multiplikationen umgerechnet werden können, mag dies durchaus akzeptabel sein. Bei Größenarten, die aber auch logarithmische Einheiten umfassen, wäre diese Vorgehensweise bei der Umrechnung von einer logarithmischen Einheit (z.B. „dBm“) in eine andere logarithmische Einheit (z.B. „dbµW“) ineffizient. Hier müsste ein Wert in „dBm“ über eine Exponentialfunktion zunächst in die SI-Einheit Watt umgerechnet werden, um anschließend über eine Logarithmusfunktion in die Einheit „dBµW“ umgerechnet zu werden. Da sich die beiden logarithmischen Einheiten aber lediglich durch einen konstanten Summenwert

unterscheiden, wäre es weit sinnvoller, den Wert in „dBm“ unmittelbar in „dBµW“ zu konvertieren.

Aus diesem Grund wird ein Modell für die Umrechnung der Einheiten innerhalb einer Größenart gewählt, in der für jede Einheit eine Umrechnungsfunktion in jede andere Einheit derselben Größenart definiert wird. Hierzu muss jede Einheit eine Tabelle mit Umrechnungsfunktionen besitzen, die so viele Elemente erhält, wie Einheiten innerhalb der Größenart definiert sind. Bei der Definition der Einheiten werden nur die Parameter mit angegeben, die notwendig sind, um die Einheit einer bestimmten Größenart zuzuweisen, die Einheit innerhalb der Größenart eindeutig zu identifizieren, sie als lineare oder logarithmische Einheit auszuweisen und um sowohl die *FctConvertIntoSI*- als auch die *FctConvertFromSI*-Umrechnungsfunktion bestimmen zu können. Wurden alle Einheiten einer Größenart angelegt, kann eine Initialisierungsmethode der Klasse *CPhysSize* automatisch alle direkten Umrechnungsfunktionen zwischen den Einheiten der Größenart ermitteln und in die Konvertierungs-Tabellen der Einheiten eintragen.

Zur Festlegung einer Umrechnungsfunktion muss eine geeignete Struktur gewählt werden, mit der es möglich ist:

- Eine lineare Einheit in eine andere lineare Einheit umzurechnen (z.B. mW -> µW)
- Eine lineare Einheit in eine logarithmische Einheit umzurechnen (z.B. mW -> dBµW)
- Eine logarithmische Einheit in eine lineare Einheit umzurechnen (z.B. dBm -> µW)
- Eine logarithmische Einheit in eine andere logarithmische Einheit umzurechnen (z.B. dBm -> dBµW)

Wie nachfolgende Erläuterungen noch zeigen werden (und ein guter Mathematiker vielleicht auch herleiten kann), lassen sich über drei Arten von Funktionen mit jeweils zwei Freiheitsgraden alle Einheiten innerhalb einer Größenart umrechnen. Diese drei Funktionstypen sind:

1. **Lin**eare Geradengleichung: $\mathbf{y} = \mathbf{m}x + \mathbf{t}$
2. **Log**arithmusfunktion: $\mathbf{y} = \mathbf{m} * \log_{10}(\mathbf{x}) + \mathbf{t}$
3. **Exp**onentialfunktion: $\mathbf{y} = 10^{((\mathbf{x}+\mathbf{t})/\mathbf{m})}$

Damit die Umrechnungsfunktion noch eindeutig der Quell- und der Ziel-Einheit zugeordnet werden kann, zwischen denen umzurechnen ist, werden in die Struktur zusätzliche Referenzen auf diese beiden Einheiten mit aufgenommen. Ferner wollen wir für Debugging und Dokumentationszwecke noch einen String mit aufnehmen, der die Umrechnungsfunktion in leicht lesbarer, mathematischer Form wiedergibt.

In nachfolgenden Ausführungen werden die direkten Umrechnungsfunktionen zwischen Einheiten derselben Größenart hergeleitet und in eine Form mit zwei konstanten Werten für „m“ und „t“ gebracht.

Umrechnung zwischen linearen Einheiten

Lineare Einheiten lassen sich durch eine einfache Multiplikation in und aus der SI-Einheit umrechnen.

FctConvertIntoSI	*FctConvertFromSI*
$\mathbf{y}_{SI} = \mathbf{m}_{SrcIntoSI} * \mathbf{x}_{Src}$	$\mathbf{y}_{Dst} = \mathbf{m}_{DstFromSI} * \mathbf{x}_{SI}$

1. Src Into SI

$\mathbf{y}_{SI} = \mathbf{m}_{SrcIntoSI} * \mathbf{x}_{Src}$

2. Dst From SI

$\mathbf{y}_{Dst} = \mathbf{m}_{DstFromSI} * \mathbf{x}_{SI}$

3. (1) in (2)

$\mathbf{y}_{Dst} = \mathbf{m}_{DstFromSI} * (\mathbf{m}_{SrcIntoSI} * \mathbf{x}_{Src})$

4. Umformen in Geradengleichung $\mathbf{y} = \mathbf{m}x + \mathbf{t}$

$\mathbf{y}_{Dst} = \mathbf{m}_{Res} * \mathbf{x}_{Src}$

$\mathbf{Fct}_{Res}$	$= \mathbf{Lin}$
$\mathbf{m}_{Res}$	$= \mathbf{m}_{DstFromSI} * \mathbf{m}_{SrcIntoSI}$
$\mathbf{t}_{Res}$	$= 0$

Umrechnung einer linearen in eine logarithmische Einheit

Eine lineare Einheit lässt sich durch eine einfache Multiplikation in die SI-Einheit umrechnen. Um die SI-Einheit in die logarithmische Einheit zu überführen, muss eine Logartihmusfunktion angewendet werden:

$\mathbf{L}/B = \log_{10}(\mathbf{P}/\mathbf{P}_{Rel})$ bzw.
$\mathbf{L}/dB = 10*\log_{10}(\mathbf{P}/\mathbf{P}_{Rel})$

Ausgehend von Feldgrößen ergibt sich das Pegelmaß zu:

$\mathbf{L}/B = 2*\log_{10}(\mathbf{x}/\mathbf{x}_{Rel})$ bzw.
$\mathbf{L}/dB = 20*\log_{10}(\mathbf{x}/\mathbf{x}_{Rel})$

$\mathbf{P}_{Rel}$ und $\mathbf{x}_{Rel}$ sind die Bezugsgrößen, die durch das „Anhängsel“ an die Einheit (z.B. „m“ bei „dBm“ für $\mathbf{P}_{Rel}$ = 1mW, „u“ bei „dBu“ für $\mathbf{x}_{Rel} = \sqrt{(600\Omega*0{,}001W)} \approx 0{,}7746$ V) festgelegt werden.

Um einen Wert $\mathbf{P}$ aus der SI-Einheit Watt in dBm mit der Bezugsgröße $\mathbf{P}_{Rel}$ = 1,0 mW umzurechnen, wäre folgende Rechenoperation notwendig:

$\mathbf{L}/dBm = 10*\log_{10}((\mathbf{P}/W) / 1mW) = 10*\log_{10}(10^3*\mathbf{P}) = 10*\log_{10}(\mathbf{P}) + 10*\log_{10}(10^3)$
$\mathbf{L}/dBm = 10*\log_{10}(\mathbf{P}) + 30{,}0$

Oder anders ausgedrückt:

$\mathbf{y}_{Dst} = \mathbf{m}_{DstFromSI} * \log10(\mathbf{x}_{SI}) + \mathbf{t}_{DstFromSI}$

mit:

$\mathbf{m}_{DstFromSI} = 10$
$\mathbf{t}_{DstFromSI} = 10*\log_{10}(\mathbf{P}_{Rel})$

Damit haben wir also folgende zwei Umrechnungsfunktionen zu berücksichtigen:

FctConvertIntoSI	*FctConvertFromSI*
$\mathbf{y}_{SI} = \mathbf{m}_{SrcIntoSI} * \mathbf{x}_{Src}$	$\mathbf{y}_{Dst} = \mathbf{m}_{DstFromSI} * \log10(\mathbf{x}_{SI}) + \mathbf{t}_{DstFromSI}$

1. Src Into SI

$\mathbf{y}_{SI} = \mathbf{m}_{SrcIntoSI} * \mathbf{x}_{Src}$

2. Dst From SI

$\mathbf{y}_{Dst} = \mathbf{m}_{DstFromSI} * \log10(\mathbf{x}_{SI}) + \mathbf{t}_{DstFromSI}$

3. (1) in (2)

$\mathbf{y}_{Dst} = \mathbf{m}_{DstFromSI} * \log10(\mathbf{m}_{SrcIntoSI} * \mathbf{x}_{Src}) + \mathbf{t}_{DstFromSI}$

4. Umformen in Logarithmus-Funktion mit $\mathbf{y} = \mathbf{m}*\log_{10}(\mathbf{x}) + \mathbf{t}$

$\mathbf{y}_{Dst} = \mathbf{m}_{DstFromSI} * \log10(\mathbf{x}_{Src}) + \mathbf{m}_{DstFromSI} * \log10(\mathbf{m}_{SrcIntoSI}) + \mathbf{t}_{DstFromSI}$

$\mathbf{Fct}_{Res}$	= **Log**
$\mathbf{m}_{Res}$	$= \mathbf{m}_{DstFromSI}$
$\mathbf{t}_{Res}$	$= \mathbf{m}_{DstFromSI} * \log_{10}(\mathbf{m}_{SrcIntoSI}) + \mathbf{t}_{DstFromSI}$

Umrechnung einer logarithmischen in eine lineare Einheit

Eine lineare Einheit lässt sich durch einfache Multiplikation aus der SI-Einheit umrechnen. Um eine logarithmische Einheit in die SI-Einheit zu überführen, muss die Umkehrfunktion der Logarithmusfunktion – also die Exponentialfunktion - angewendet werden:

Da die Umrechnungsfunktion *FctConvertFromSI* für logarithmische Einheiten in der Form

$$y_{Dst} = \mathbf{m}_{DstFromSI} * \log_{10}(x_{SI}) + \mathbf{t}_{DstFromSI}$$

festgelegt wurde, lautet die Umrechnungsfunktion für *FctConvertIntoSI* Unit wie folgt:

$$x/_{SI} = 10^{((y_{Src} - \mathbf{t}SrcFromSI) / \mathbf{m}SrcFromSI)}$$

$\mathbf{Fct}_{IntoSI}$	= **Exp**
$\mathbf{m}_{IntoSI}$	= $\mathbf{m}_{DstFromSI}$
$\mathbf{t}_{IntoSI}$	= -$\mathbf{t}_{DstFromSI}$

Damit haben wir also folgende zwei Umrechnungsfunktionen zu berücksichtigen:

FctConvertIntoSI	*FctConvertFromSI*
$y/_{SI} = 10^{((x_{Src} + \mathbf{t}SrcIntoSI) / \mathbf{m}SrcIntoSI)}$	$y_{Dst} = \mathbf{m}_{DstFromSI} * x_{SI}$

Anmerkung: Innerhalb *FctConvertIntoSI* ist der Operand $\mathbf{t}_{SrcIntoSI}$ negativ, damit ist die anzuwendende Funktion „+“

1. Src Into SI

$$y/_{SI} = 10^{((x_{Src} + \mathbf{t}SrcIntoSI) / \mathbf{m}SrcIntoSI)}$$

2. Dst From SI

$$y_{Dst} = \mathbf{m}_{DstFromSI} * x_{SI}$$

3. (1) in (2)

$$y_{Dst} = \mathbf{m}_{DstFromSI} * 10^{((x_{Src} + \mathbf{t}SrcIntoSI) / \mathbf{m}SrcIntoSI)}$$

4. Umformen in Exponential-Funktion mit $y = 10^{((x+t)/m)}$

da $a*10^b = 10^{\log10(a)}*10^b = 10^{(\log10(a)+b)}$ folgt

$$y_{Dst} = 10^{(\log10(\mathbf{m}DstFromSI) + (x_{Src} + \mathbf{t}SrcIntoSI) / \mathbf{m}SrcIntoSI))}$$
$$y_{Dst} = 10^{(\log10(\mathbf{m}DstFromSI) + x_{Src}/ \mathbf{m}SrcIntoSI + \mathbf{t}SrcIntoSI/\mathbf{m}SrcIntoSI)}$$
$$y_{Dst} = 10^{((x_{Src} + \mathbf{m}SrcIntoSI*\log10(\mathbf{m}DstFromSI) + \mathbf{t}SrcIntoSI) / \mathbf{m}SrcIntoSI)}$$

$\mathbf{Fct}_{Res}$	= **Exp**
$\mathbf{m}_{Res}$	= $\mathbf{m}_{SrcIntoSI}$
$\mathbf{t}_{Res}$	= $\mathbf{m}_{SrcIntoSI} * \log_{10}(\mathbf{m}_{DstFromSI}) + \mathbf{t}_{SrcIntoSI}$

Umrechnung zwischen logarithmischen Einheiten

Die Umrechnungsfunktionen *FctConvertIntoSI* und *FctConvertFromSI* wurden für logarithmische Einheiten bereits wie folgt hergeleitet:

FctConvertIntoSI	*FctConvertFromSI*
$\mathbf{y}/_{SI} = 10^{((\mathbf{x}Src + \mathbf{t}SrcIntoSI) / \mathbf{m}SrcIntoSI)}$	$\mathbf{y}_{Dst} = \mathbf{m}_{DstFromSI} * \log10(\mathbf{x}_{SI}) + \mathbf{t}_{DstFromSI}$

Anmerkung: Innerhalb *FctConvertIntoSI* ist der Operand $\mathbf{t}_{SrcIntoSI}$ negativ, damit ist die anzuwendende Funktion „+“

5. Src Into SI

$\mathbf{y}/_{SI} = 10^{((\mathbf{x}Src + \mathbf{t}SrcIntoSI) / \mathbf{m}SrcIntoSI)}$

6. Dst From SI

$\mathbf{y}_{Dst} = \mathbf{m}_{DstFromSI} * \log10(\mathbf{x}_{SI}) + \mathbf{t}_{DstFromSI}$

7. (1) in (2)

$\mathbf{y}_{Dst} = \mathbf{m}_{DstFromSI} * \log10(^{10((\mathbf{x}Src + \mathbf{t}SrcIntoSI) / \mathbf{m}SrcIntoSI)}) + \mathbf{t}_{DstFromSI}$

8. Umformen in Geradengleichung mit $\mathbf{y} = \mathbf{m}\mathbf{x} + \mathbf{t}$

da: $\log_{10}(10^{\mathbf{x}}) = \mathbf{x}$

folgt:

$\mathbf{y}_{Dst} = \mathbf{m}_{DstFromSI} * (\mathbf{x}_{Src} + \mathbf{t}_{SrcIntoSI}) / \mathbf{m}_{SrcIntoSI} + \mathbf{t}_{DstFromSI}$
$\mathbf{y}_{Dst} = \mathbf{m}_{DstFromSI}/\mathbf{m}_{SrcIntoSI} * \mathbf{x}_{Src} + (\mathbf{m}_{DstFromSI}/\mathbf{m}_{SrcIntoSI}) * \mathbf{t}_{SrcIntoSI} + \mathbf{t}_{DstFromSI}$

$\mathbf{Fct}_{Res}$	= **Lin**
$\mathbf{m}_{Res}$	$= \mathbf{m}_{DstFromSI} / \mathbf{m}_{SrcIntoSI}$
$\mathbf{t}_{Res}$	$= \mathbf{m}_{DstFromSI} / \mathbf{m}_{SrcIntoSI} * \mathbf{t}_{SrcIntoSI} + \mathbf{t}_{DstFromSI}$

2.3. Umrechnungen zwischen Einheiten unterschiedlicher Größenarten

Ein weiterer, häufiger Anwendungsfall ist die direkte Umrechnung einer physikalischen Größe einer Größenart in eine physikalische Größe einer anderen Größenart. So will man sich z.B. in der Nachrichtentechnik einen Leistungspegel in „dBm“ häufig auch in „Volt“ bzw. „dBV“ oder „dBµV“ anzeigen lassen. Dies ist dann möglich, wenn man sich sog. Referenzwerte bedient. So wäre z.B. in der Hochfrequenztechnik der Referenzwert der Widerstand 50 Ω.

Betrachtet man die Dimensionen der abgeleiteten Einheiten, so wird man sehen, dass man aus den Dimensionen unmittelbar die Umrechnungsfunktionen ableiten könnte. Hierzu noch einmal ein Auszug aus der Tabelle mit der Unterteilung der physikalischen Einheiten in Wissensgebiete:

Wissensgebiet	Größenart	Physikalische Größe	Formelzeichen	Dimension	SI-Einheit
Elektrizität und Magnetismus	El. Stromstärke	El. Stromstärke	***I***	**I**	Ampere (A)
	El. Spannung	El. Spannung	***U***	$\mathbf{ML^2/T^3I^1}$	Volt (V)
	El. Widerstand	Ohmscher Widerstand	***R***	$\mathbf{ML^2/T^3I^2}$	Ohm (Ω)
	Leistung	Wirkleistung	***P***	$\mathbf{ML^2/T^3}$	Watt (W)
	…	…	**…**	**…**	…

Da U = ML²/(T³I) und P = ML²/T³ folgt U = P/I
Da R = ML²/(T³I²) und P = ML²/T³ folgt R = P/I² bzw. I = √(P/R)
Somit folgt: U = P/√(P/R) = √(P*R)

Rein theoretisch könnte man also einen ausgefeilten Algorithmus implementieren, der automatisch anhand der Dimensionen die Umrechnungsfunktion ermittelt, um die Leistung P in die Spannung U mit Hilfe eines Referenzwiderstands R umzurechnen. Aber auch wenn der Algorithmus noch so gut implementiert wäre, würde man sich nicht tragbare Laufzeit-Einbußen einhandeln, müsste man die Umrechnungsfunktion bei jeder Konvertierung von neuem ermitteln. Das dürfte also nur einmal geschehen - sinnvollerweise während der Initialisierungsphase des Systems. Aber zu diesem Zeitpunkt kann das System selbst nicht entscheiden, welche Umrechnungen zur Laufzeit benötigt werden und es wäre ein Unding, für jede theoretisch mögliche Umrechnung eine entsprechende Funktion zu generieren.

Deshalb wurde hier ein anderer Weg beschritten. Betrachtet man obige Dimensions-Gleichungen, so sind diese weit weniger geläufig, als das Ohmsche Gesetz oder die Gleichung zur Berechnung der Leistung als Produkt aus Spannung und Strom:

R = U/I (damit gilt: U = RI bzw. I = U/R)
P = U*I (damit gilt: U = P/I bzw. I = P/U)

Ist nun R bekannt und muss U in Abhängigkeit von P bestimmt werden, ist I = U/R in die Gleichung U = P/I einzusetzen. Ist umgekehrt die Leistung P in Abhängigkeit von U bei bekanntem R zu berechnen, wäre I = U/R in die Gleichung P = U*I einzusetzen:

U = P/(U/R) -> U² = P*R -> U = √(P*R)
P = U²/R

Aber weder die Funktion U = √(P*R) noch die Funktion P = U²/R könnte auf eine der drei vordefinierten Funktionstypen Lin, Log oder Exp mit den Operanden **m** und **t** umgewandelt werden. Um praktisch jede geläufige Einheitenumrechnung zu ermöglichen, könnte deshalb ein benutzerdefinierter Funktionstyp eingeführt werden, dessen Signatur wie folgt aussehen könnte:

```
typedef double (*TFctConvertUser1)(
    const CPhysUnit&  i_physUnitSrc,
    double            i_fVal,
    const CPhysUnit&  i_physUnitDst,
    double            m_fM,
    double            m_fT,
    CPhysUnit*        i_pPhysUnitRef,
    double            i_fValRef );
```

Bei der Umsetzung des Anwendungsfalls „Umrechnung zwischen Einheiten unterschiedlicher Größenart„ hat sich jedoch eine praktikablere Lösung herauskristallisiert, in der die vordefinierten Funktionstypen **Lin**, **Log** und **Exp** um weitere Funktionstypen ergänzt wurden, die gleichzeitig mathematische Berechnungsvorschriften enthalten und außerdem ein weiterer Freiheitsgrad „**k**“ eingeführt wurde. Es hat sich nämlich gezeigt, dass nur ein geringer Satz von Umrechnungsfunktionen nötig ist, um Einheiten unterschiedlicher Größenarten umzurechnen und die anzuwendenden mathematischen Gleichungen und Konstanten weitgehend automatisch ermittelt werden können.

Da die tatsächlich implementierte Art und Weise der Definition der Umrechnungs-Funktionen sich lediglich durch die Anwendung einer erweiterten Type Definition gegenüber dem Ansatz mit den benutzerdefinierten Funktionstypen unterscheidet sowie einer zusätzlichen Variablen „k“, wird im Folgenden der Ansatz mit den benutzerdefinierten Funktionstypen weiter erläutert. Außerdem bieten die benutzerdefinierten Funktionstypen einen Freiheitsgrad, den wir uns auch noch für die Zukunft offen halten wollen, wenn es darum gehen solle, „exotische“ Einheiten-Umrechnungen vornehmen zu müssen.

Für unser Modell der physikalischen Einheiten würde ein benutzerdefinierter Funktionstyp bedeuten, dass wir zusätzlich zu den Umrechnungs-Funktionen innerhalb der Größenart noch eine Tabelle für Umrechnungs-Funktionen in eine beliebige andere Größenart vorsehen müssen. Ein Automatismus für diese Umrechnungs-Funktionen könnte insofern realisiert werden, als dass es ausreichend ist, Umrechnungs-Funktion zwischen den SI-Einheiten der verschiedenen Größenarten festzulegen, in die zu konvertieren ist. Sind logarithmische Einheiten im Spiel, ist der Umweg über die SI-Einheit wiederum nicht besonders effizient, so dass es sich empfiehlt, z.B. bei den Umrechnungen von dBW in dBV oder dBµW in dBu jeweils direkte Umrechnungs-Funktionen zu implementieren und den Einheiten zuzuweisen.

Anmerkung: Der Funktionstyp wurde bewusst mit „User1“ festgelegt, um darauf hinzuweisen, dass auch weitere Prototypen denkbar sind und implementiert werden könnten, falls komplexere Umrechnungsfunktionen notwendig sind.

Über die Methode „*addFctConvertExternal*“ wird für eine Einheit eine solche Umrechnungs-Funktion festgelegt. Dabei werden die Parameter „*physUnitDst*“, „*M*“, „*T*“, und „*physUnitRef*“ beim Hinzufügen der „externen“ Umrechnungsfunktion definiert und

zur Laufzeit an die Konvertierungsmethode übergeben. Die Klasse *CPhysSize* wird um eine virtuelle Methode „*getRefVal*“ erweitert, die per Default eine Exception wirft. Diese muss überschrieben werden, soll z.B. für elektrische Widerstände der Referenzwert 50 Ohm zurückgegeben werden. Der so ermittelte Referenzwert wird ebenfalls zur Laufzeit an die Konvertierungsmethode übergeben.

Die eigentliche Einheiten-Konvertierung wird durch einen Methodenaufruf „*CPhysUnit::convertValue*“ durchgeführt. Da es in manchen Anwendungsfällen vorkommt, dass sich der Referenzwert dynamisch zur Laufzeit ändert bzw. nicht der „Default“-Referenzwert für die Umrechnungsfunktion verwendet werden soll, wird eine weitere, polymorphe „*convertValue*“ Methode implementiert, an die zusätzlich der gewünschte Referenzwert übergeben wird. In diesem Fall wird nicht der mittels „*getRefVal*“ ermittelte Default-Referenzwert sondern der explizit bei Aufruf von „*convertValue*“ übergebene Referenzwert an die benutzerdefinierte Umrechnungsfunktion weitergegeben. Dadurch wird es sogar möglich, ohne einen vordefinierten Referenzwert Einheiten zwischen verschiedenen Größenarten umzurechnen.

Um Einheiten aus einer Größenart in eine anderen Größenart zu konvertieren werden benutzerdefinierte Umrechnungsfunktionen benötigt, die über den Aufruf „*addFctConvertExternal*“ an die zu konvertierenden Einheiten zu übergeben sind.

Betrachten wir uns hierzu die Umrechnung einer elektrischen Spannung U/V in die elektrische Leistung P/W (und umgekehrt) bei bekanntem Referenzwiderstand R/ Ω = 50 nach folgenden Formeln:

$U = \sqrt{(P*R)}$
$P = U^2/R$

Bei bekanntem Referenzwiderstand würden sich aber auch ebenso Spannung in Strom, Strom in Spannung, Leistung in Strom sowie Strom in Leistung nach folgenden Formeln umrechnen lassen:

$U = I*R$
$I = U/R$
$I = \sqrt{(P/R)}$
$P = I^2*R$

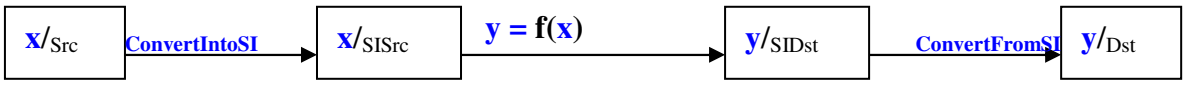
Diese Umrechnungen könnten Funktionen wie „*Sqrt(xMULref)*“, „*SQRxDIVref*“, „xMULref“, „xDIVref“, „*SQRT(xDIVref)*“ und „*SQRxMULref*“ durchführen. Da man Klammern bei der Deklaration von Methoden nicht verwenden kann, die Setzung der Klammern aber entscheidend ist für die Bestimmung der durchgeführten Rechenoperation, werden die Klammern bei der Deklaration der Funktion durch einen Unterstrich ersetzt. Ferner erhalten die Umrechnungsfunktionen noch einen Vorspann (*fctConvertUser1*), um sie als Einheiten-Umrechnungsfunktionen kenntlich zu machen. Der Vorspann wird ebenfalls durch einen Unterstrich von der mathematischen Funktion getrennt. Um außerdem innerhalb der Bezeichnung für die mathematische Operation im Methodennamen die Anweisungen für Rechenoperationen von den Operatoren unterscheiden zu können, werden Rechenoperation in Großbuchstaben, die Operatoren in Kleinbuchstaben geschrieben.

Zur Umrechnung der elektrischen Größenwerte bei bekanntem Referenzwiderstand werden deshalb folgende Umrechnungsfunktionen deklariert:

```
double fctConvertUser1_xMULref_( .. )       // y=x*r
double fctConvertUser1_xDIVref( .. )        // y=x/r
double fctConvertUser1_SQRT_xMULref_( .. ) // y=√(x*r)
double fctConvertUser1_SQRxDIVref( .. )     // y=x²/r
double fctConvertUser1_SQRT_xDIVref_( .. ) // y=√(x/r)
double fctConvertUser1_SQRxMULref( .. )     // y=x²*r
```

Die Signaturen der Umrechnungsfunktionen entsprechen dem Prototype, wie er unter *TFctConvertUser1* deklariert wurde. Es ist leicht einzusehen, dass diese Umrechnungsfunktionen nicht nur bei der Konvertierung innerhalb eines Wissensgebiets, sondern auch bei der Umrechnung von Einheiten in andere Wissensgebiete Verwendung finden können, wie z.B. bei der Umrechnung einer Wegstrecke (Geometrie) in eine Geschwindigkeit (Kinematik) bei vorgegebener Zeit etc.

In den Formeln sind die Größenwerte jeweils normiert auf die SI-Einheiten (Volt, Watt, Ampere und Ohm) einzutragen. Die Umrechnungsfunktionen müssen also die Werte zunächst in die SI-Einheit der Quellgrößenart, danach durch Anwendung der Formel in die andere Größenart umrechnen und schließlich den Wert aus der SI-Einheit der Zielgrößenart in die gewünschte Zieleinheit umrechnen. Da die Klasse *CPhysUnit* sowohl eine Umrechnungsvorschrift enthält, um die Einheit in die SI-Einheit als auch die Einheit aus der SI-Einheit umzurechnen, ist die Implementierung der Methode denkbar einfach und folgt folgendem Muster:



Das würde aber bedeuten, dass auch, um eine Leistung in dBm nach dBV umzurechnen, der Wert in dBm zuerst in Watt umgerechnet, dann in Volt konvertiert und anschließend in die Zieleinheit dBV umgerechnet werden müsste. Das ist nicht optimal, da sowohl eine Logarithmus- als auch eine Exponential-Funktion beteiligt wären, die Rechenzeit kosten. Dabei unterscheiden sich logarithmische Spannungs- und logarithmische Leistungswerte lediglich um einen Faktor und einen konstanten Summanden und könnten über einfache, lineare Geradengleichungen umgerechnet werden. Diese Geradengleichungen sollen im Folgenden hergeleitet werden. Der Referenzwert wird in den Gleichungen mit einem kleinen „**r**“ bezeichnet.

1. Fall: y = x*r

(1) $x_{SI} = 10^{((x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})}$

(2) $y_{SI} = x_{SI} * r_{SI}$

(3) $y_{Dst} = m_{DstFromSI} * \log_{10}(y_{SI}) + t_{DstFromSI}$

(1) in (2): $y_{SI} = 10^{((x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})} * r_{SI}$

(2) in (3): $y_{Dst} = m_{DstFromSI} * \log_{10}(10^{((x_{Src}+t_{SrcIntoSI}) / m_{SrcIntoSI})} * r_{SI}) + t_{DstFromSI}$

$y_{Dst} = m_{DstFromSI} * \log_{10}(10^{((x_{Src}+t_{SrcIntoSI}) / m_{SrcIntoSI})} + m_{DstFromSI} * \log_{10}(r_{SI}) + t_{DstFSI}$

$y_{Dst} = (m_{DstFromSI}/m_{SrcIntoSI}) * (x_{Src}+t_{SrcIntoSI}) + m_{DstFromSI} * \log_{10}(r_{SI}) + t_{DstFromSI}$

Umformung in Geradengleichung mit $y = mx + t$:

$y_{Dst} = (m_{DstFSI}/m_{SrcIntoSI}) * x_{Src} + (m_{DstFSI}/m_{SrcIntoSI}) * t_{SrcIntoSI} + m_{DstFSI} * \log_{10}(r_{SI}) + t_{DstFSI}$

Fct_{Res}	= Lin
m_{Res}	$= m_{DstFromSI} / m_{SrcIntoSI}$
t_{Res}	$= m_{DstFromSI} / m_{SrcIntoSI} * t_{SrcIntoSI} + m_{DstFromSI} * \log_{10}(r_{SI}) + t_{DstFromSI}$

2. Fall: y = x/r

(1) $x_{SI} = 10^{((x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})}$

(2) $y_{SI} = x_{SI} / r_{SI}$

(3) $y_{Dst} = m_{DstFromSI} * \log_{10}(y_{SI}) + t_{DstFromSI}$

(1) in (2): $y_{SI} = 10^{((x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})} / r_{SI}$

(2) in (3): $y_{Dst} = m_{DstFromSI} * \log_{10}(10^{((x_{Src}+t_{SrcIntoSI}) / m_{SrcIntoSI})} / r_{SI}) + t_{DstFromSI}$

$y_{Dst} = m_{DstFromSI} * \log_{10}(10^{((x_{Src}+t_{SrcIntoSI}) / m_{SrcIntoSI})} - m_{DstFromSI} * \log_{10}(r_{SI}) + t_{DstFSI}$

$y_{Dst} = (m_{DstFromSI}/m_{SrcIntoSI}) * (x_{Src}+t_{SrcIntoSI}) - m_{DstFromSI} * \log_{10}(r_{SI}) + t_{DstFromSI}$

Umformung in Geradengleichung mit $y = mx + t$:

$y_{Dst} = (m_{DstFSI}/m_{SrcIntoSI}) * x_{Src} + (m_{DstFSI}/m_{SrcIntoSI}) * t_{SrcIntoSI} - m_{DstFSI} * \log_{10}(r_{SI}) + t_{DstFSI}$

Fct_{Res}	= Lin
m_{Res}	$= m_{DstFromSI} / m_{SrcIntoSI}$
t_{Res}	$= m_{DstFromSI} / m_{SrcIntoSI} * t_{SrcIntoSI} - m_{DstFromSI} * \log_{10}(r_{SI}) + t_{DstFromSI}$

3. Fall: $y = x^2/r$

(1) $x_{SI} = 10^{((x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})}$

(2) $y_{SI} = x_{SI}^2/r_{SI}$

(3) $y_{Dst} = m_{DstFromSI} * \log_{10}(y_{SI}) + t_{DstFromSI}$

(1) in (2): $y_{SI} = 10^{(2*(x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})} / r_{SI}$

(2) in (3): $y_{Dst} = m_{DstFromSI}*\log_{10}(10^{(2*(x_{Src}+t_{SrcIntoSI}) / m_{SrcIntoSI})} / r_{SI}) + t_{DstFromSI}$

$y_{Dst} = m_{DstFromSI}*\log_{10}(10^{(2*(x_{Src}+t_{SrcIntoSI}) / m_{SrcIntoSI})} - m_{DstFromSI}*\log_{10}(r_{SI}) + t_{DstFSI}$

$y_{Dst} = 2*(m_{DstFromSI}/m_{SrcIntoSI})*(x_{Src}+t_{SrcIntoSI}) - m_{DstFromSI}*\log_{10}(r_{SI}) + t_{DstFromSI}$

Umformung in Geradengleichung mit $y = mx + t$:

$y_{Dst} = 2(m_{DstFSI}/m_{SrcIntoSI})*x_{Src} + 2(m_{DstFSI}/m_{SrcIntoSI})*t_{SrcIntoSI} - m_{DstFSI}*\log_{10}(r_{SI}) + t_{DstFSI}$

Fct_{Res}	= Lin
m_{Res}	$= 2 * m_{DstFromSI} / m_{SrcIntoSI}$
t_{Res}	$= 2 * m_{DstFromSI} / m_{SrcIntoSI} * t_{SrcIntoSI} - m_{DstFromSI}*\log_{10}(r_{SI}) + t_{DstFromSI}$

4. Fall: $y = \sqrt{(x*r)}$

(1) $x_{SI} = 10^{((x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})}$

(2) $y_{SI} = \sqrt{(x_{SI}*r_{SI})} = x_{SI}^{½}*r_{SI}^{½}$

(3) $y_{Dst} = m_{DstFromSI} * \log_{10}(y_{SI}) + t_{DstFromSI}$

(1) in (2): $y_{SI} = (10^{(½*(x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})}) * r_{SI}^{½}$

(2) in (3): $y_{Dst} = m_{DstFromSI} * \log_{10}(10^{(½*(x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})} * r_{SI}^{½}) + t_{DstFromSI}$

$y_{Dst} = m_{DstFSI}*\log_{10}(10^{(½*(x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})} + m_{DstFSI}*\log_{10}(r_{SI}^{½}) + t_{DstFSI}$

$y_{Dst} = ½*(m_{DstFSI}/m_{SrcIntoSI})*(x_{Src}+t_{SrcIntoSI}) + ½*m_{DstFSI}*\log_{10}(r_{SI}) + t_{DstFSI}$

Umformung in Geradengleichung mit $y = mx + t$:

$y_{Dst} = ½*(m_{DstFSI}/m_{SrcInSI})*x_{Src} + ½*(m_{DstFSI}/m_{SrcInSI})*t_{SrcInSI} + ½*m_{DstFSI}*\log_{10}(r_{SI}) + t_{DstFSI}$

Fct_{Res}	= Lin
m_{Res}	$= ½ * m_{DstFromSI} / m_{SrcIntoSI}$
t_{Res}	$= ½ * m_{DstFromSI} / m_{SrcIntoSI} * t_{SrcIntoSI} + ½*m_{DstFromSI}*\log_{10}(r_{SI}) + t_{DstFromSI}$

5. Fall: $y = x^2 * r$

(1) $x_{SI} = 10^{((x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})}$
(2) $y_{SI} = x_{SI}^2 * r_{SI}$
(3) $y_{Dst} = m_{DstFromSI} * \log_{10}(y_{SI}) + t_{DstFromSI}$

(1) in (2): $y_{SI} = 10^{(2*(x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})} * r_{SI}$
(2) in (3): $y_{Dst} = m_{DstFromSI} * \log_{10}(10^{(2*(x_{Src}+t_{SrcIntoSI}) / m_{SrcIntoSI})} * r_{SI}) + t_{DstFromSI}$
$y_{Dst} = m_{DstFromSI} * \log_{10}(10^{(2*(x_{Src}+t_{SrcIntoSI}) / m_{SrcIntoSI})} + m_{DstFromSI} * \log_{10}(r_{SI}) + t_{DstFSI}$
$y_{Dst} = 2*(m_{DstFromSI}/m_{SrcIntoSI})*(x_{Src}+t_{SrcIntoSI}) + m_{DstFromSI} * \log_{10}(r_{SI}) + t_{DstFromSI}$

Umformung in Geradengleichung mit $y = mx + t$:

$y_{Dst} = 2(m_{DstFSI}/m_{SrcIntoSI})*x_{Src} + 2(m_{DstFSI}/m_{SrcIntoSI})*t_{SrcIntoSI} + m_{DstFSI} * \log_{10}(r_{SI}) + t_{DstFSI}$

Fct_{Res}	= Lin
m_{Res}	$= 2 * m_{DstFromSI} / m_{SrcIntoSI}$
t_{Res}	$= 2 * m_{DstFromSI} / m_{SrcIntoSI} * t_{SrcIntoSI} + m_{DstFromSI} * \log_{10}(r_{SI}) + t_{DstFromSI}$

6. Fall: $y = \sqrt{(x/r)}$

(1) $x_{SI} = 10^{((x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})}$
(2) $y_{SI} = \sqrt{(x_{SI}/r_{SI})} = x_{SI}^{½} * r_{SI}^{-½}$
(3) $y_{Dst} = m_{DstFromSI} * \log_{10}(y_{SI}) + t_{DstFromSI}$

(1) in (2): $y_{SI} = (10^{(½*(x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})}) * r_{SI}^{-½}$
(2) in (3): $y_{Dst} = m_{DstFromSI} * \log_{10}(10^{(½*(x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})} * r_{SI}^{-½}) + t_{DstFromSI}$
$y_{Dst} = m_{DstFSI} * \log_{10}(10^{(½*(x_{Src} + t_{SrcIntoSI}) / m_{SrcIntoSI})} + m_{DstFSI} * \log_{10}(r_{SI}^{-½}) + t_{DstFSI}$
$y_{Dst} = ½*(m_{DstFSI}/m_{SrcIntoSI})*(x_{Src}+t_{SrcIntoSI}) - ½*m_{DstFSI} * \log_{10}(r_{SI}) + t_{DstFSI}$

Umformung in Geradengleichung mit $y = mx + t$:

$y_{Dst} = ½*(m_{DstFSI}/m_{SrcInSI})*x_{Src} + ½*(m_{DstFSI}/m_{SrcInSI})*t_{SrcInSI} - ½*m_{DstFSI} * \log_{10}(r_{SI}) + t_{DstFSI}$

Fct_{Res}	= Lin
m_{Res}	$= ½ * m_{DstFromSI} / m_{SrcIntoSI}$
t_{Res}	$= ½ * m_{DstFromSI} / m_{SrcIntoSI} * t_{SrcIntoSI} - ½*m_{DstFromSI} * \log_{10}(r_{SI}) + t_{DstFromSI}$

Schlussfolgerung

Wird eine der vordefinierten Umrechnungsfunktionen „*fctConvertUser1_<Math>*“ vom Type = User1 über den Funktionsaufruf „addFctConvertExternal“ einer Einheit zugewiesen, kann ein Automatismus implementiert werden kann, der für jede logarithmische Ausgangseinheit die Konstanten „**m**“ und „**t**“ für die anzuwendende Geradengleichung „**y = mx + t**„ automatisch ermittelt und es ist nicht notwendig – sondern vielmehr ein Fehler - für die logarithmischen Einheiten eine eigens definierte Umrechnungsfunktion zu implementieren und zuzuweisen. Es wird auch deutlich, dass diese Umrechnungsfunktionen nicht den Einheiten sondern den Größenart zuzuweisen sind, die die resultierenden Umrechnungsfunktionen ermitteln und an die entsprechenden, logarithmischen Einheiten weitergeben muss. Diese Umrechnungsfunktionen sind also nicht der Klasse *CPhysUnit* sondern der Klasse *CPhysSize* zu übergeben. Hierfür bietet die Klasse *CPhysSize* die Methode „*addFctConvertExternal*“ an.

2.4. Umrechnung in die „bestmögliche“ Einheit

Betrachten wir wieder das Beispiel, in dem der Wert „0,000 000 670 985 km“ in eine andere Einheit umgerechnet wurde – und das mit gutem Grund, denn der Wert „0,000 000 670 985 km“ lässt sich schlecht lesen, wohingegen derselbe Wert angegeben als „670,985 µm“ einfacher zu interpretieren ist. Mit anderen Worten, Mikrometer ist die „bessere“ Einheit, um den Wert wiederzugeben.

Die „bestmögliche“ Einheit wird definiert als die Einheit, mit der der Wert so wiedergegeben wird, dass nach Möglichkeit die erste Ziffer ungleich „0“ zwischen der ersten und dritten Dezimalstelle vor dem Dezimalpunkt liegt. Ist der Wert zu klein, um diese Darstellung zu ermöglichen, soll der Wert in der „kleinsten“ Einheit ausgegeben werden. Ist der Wert zu groß, soll der Wert in der „größten“ Einheit ausgegeben werden.

Das Klassenmodell für die physikalischen Einheiten soll es ermöglichen, einen Wert automatisch in eine bessere Einheit konvertieren zu lassen. Der Algorithmus könnte wie folgt formuliert werden:

1. Zunächst wird der Wert in die SI-Einheit der Größenart konvertiert.
2. Danach wird der Exponent des Wertes in der SI-Einheit ermittelt.
3. Anhand des Exponenten lässt sich über die Faktoren die Einheit bestimmen, mit der der Wert bestmöglich wiedergegeben werden kann.

Zu beachten ist, dass der Algorithmus nur für lineare Einheiten angewendet werden kann. Bei logarithmischen Einheiten kann man aber auch durchaus darauf verzichten, eine „bestmögliche“ Einheit zu finden. Denn eine Angabe als „-30 dBm“ ist wohl weder besser noch schlechter als den Wert als „-60 dBW“ oder als „0 dBµW“ auszugeben.

Um effektiv die „bestmögliche“ Einheit finden zu können, müssen die Einheiten innerhalb einer Größenart über „NextLower“ und „NextHigher“ Attribute verkettet werden. Dieser Verkettungsvorgang kann aber auch automatisch durch Auswerten der Faktoren (Operanden 1 der linearen Einheiten) geschehen.

2.5. Fehlerbehaftete Größenwerte

Wie bereits erwähnt, sind Messwerte fehlerbehaftet und Einstellwerte können nur mit einer bestimmten Genauigkeit vorgenommen werden. Da in einem Computergestützten System Zahlenwerte binär gespeichert werden (z.B. im IEEE-Format), können Nachkommawerte (wie z.B. der Wert 0.1) nicht exakt im Rechner gespeichert werden, sondern sind durch die maximal für die Mantisse zur Verfügung stehenden Bits begrenzt. Und da der Zahlenwert für die Messunsicherheit auf wenige Stellen (meist nur eine) begrenzt ist und nicht wieder selbst mit einer Ungenauigkeit behaftet sein soll, würde es sich anbieten, für die Angabe der Genauigkeit eines Wertes ein anderes Zahlenformat zu verwenden. So könnte es vorteilhafter sein, die Genauigkeit eines Wertes in einen ganzzahligen Wert mit zugehörigem Exponenten aufzuspalten. Zudem könnte dieses Format die Bestimmung der gültigen Stellen eines Größenwertes erleichtern und beschleunigen.

So würde z.B. die Genauigkeit ±0.23 intern als 23e-1 abgelegt, also durch die zwei ganzzahligen Zahlenwerte 23 und -1.

Problem bei diesem Format ist die Behandlung bei mathematischen Operationen wie z.B. „(123,4 km ± 0,5) km + 100000*(56,7 mm ± 0,1 mm)“. In diesem Fall könnte es zu einem Zahlbereichsüberlauf führen, da die Anzahl der Stellen für die Mantisse erweitert werden müsste. Darüberhinaus würde dieses Format auch eine Addition, Subtraktion, Multiplikation und Division physikalischer Werte erschweren, bei denen die Fehler ebenfalls entsprechend zu addieren oder zu multiplizieren sind.

Aus diesem Grunde wird auch zur Angabe der Genauigkeit ein double Wert gespeichert. Und dabei ist es auch nicht erforderlich, die Anzahl der tatsächlich gültigen Stellen mitzuführen, denn es macht keinen Unterschied, ob eine Genauigkeit mit 0,10 Meter oder nur als 0,1 Meter angegeben wird. Wird die Genauigkeit mit 0,12345 Meter angegeben, wird der Wert mit zwei unsicheren Stellen angegeben, egal wie viele Stellen die Genauigkeit hat (erste Ziffer der Genauigkeitsangabe ist entscheidend). Double Werte lassen sich (beinahe) ohne Gefahr eines Zahlbereichsüberlaufs addieren. Z.B. liefert die mathematische Operation

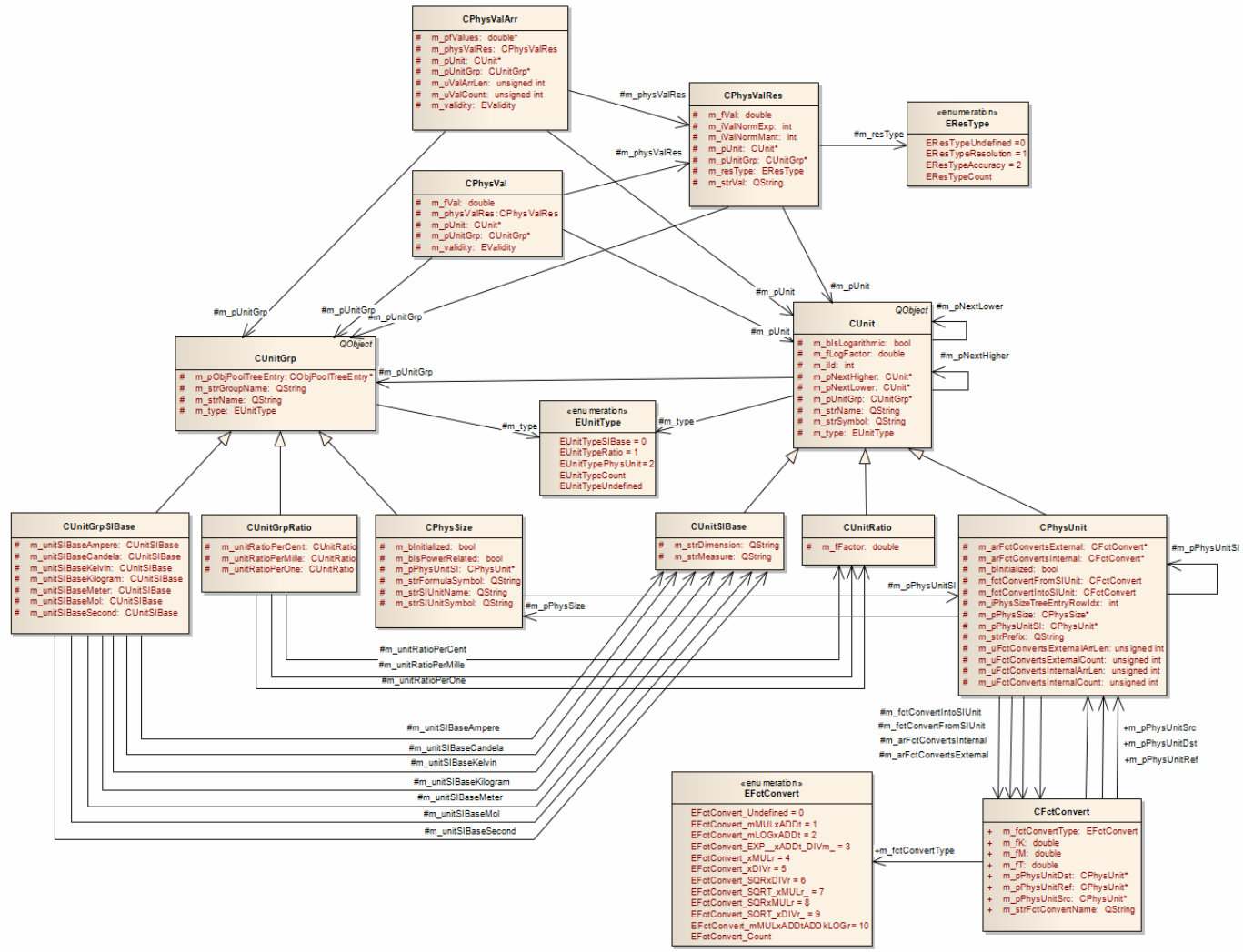
(123,4 km ± 0,5) km + 100000*(56,7 mm ± 0,1 mm)
das Ergebnis: (123,4 km ± 0,5) km + (5,670 km ± 10 m) = (129,1 km ± 0,501) km

Das heißt, der Wert wird an der unsicheren Stelle nur noch mit einer Ziffer ausgegeben, da die erste Ziffer der Ungenauigkeit größer als 2 ist. Für die Anzahl der Stellen des Messwertes ist es belanglos, ob die Ungenauigkeit 0,501 oder 0,5 beträgt.

Zusätzlich zum Zahlenwert wird noch die Einheit der Genauigkeit benötigt, die entweder eine physikalische Einheit der Größenart des Messwertes entspricht oder eine Verhältnisgröße ist (wie z.B. Prozent).

2.6. Klassendiagramm

Die genauere Untersuchung der Domäne „physikalische Werte und Einheiten“ führte zu folgendem Klassenmodell:



3. Anwendungsbeispiele

3.1. Definieren der physikalischen Größenarten und Einheiten

Innerhalb der Dll's „*ZSPhysSizesElectricity*“, „*ZSPhysSizesGeometry*“ und „*ZSPhysSizesKinematics*“ wurden exemplarisch physikalische Größenarten als Instanzen der Klasse *CPhysSize* sowie den zugehörigen SI-Einheiten und einigen, von der SI-Einheit abgeleiteten Einheiten konfiguriert.

Im Folgenden sehen wir uns die Konfiguration der Größenarten und Einheiten innerhalb des Wissensgebiets „Elektrizität und Magnetismus“ näher an, so wie sie in der Dll „*ZSPhysSizesElectricity*“ implementiert wurden.

Für jede Größenart werden von der Klasse *CPhysSize* abgeleitete Klassen implementiert und die Einheiten, die in der Applikation unterstützt werden sollen, werden als Member-Variablen angelegt. Für jede Einheit wird eine Methode definiert, die eine Referenz auf die Einheit zurückgibt.

```
//*****************************************************************
class CPhysSizeVoltage : public CPhysSize
//*****************************************************************
{
public: // ctors and dtor
    CPhysSizeVoltage();
    ~CPhysSizeVoltage();
public: // instance methods
    CPhysUnit& PicoVolt();
    ..
    CPhysUnit& Volt();
    CPhysUnit& KiloVolt();
    CPhysUnit& MegaVolt();
    CPhysUnit& dBVolt();
    CPhysUnit& dBu();
    ..
    CPhysUnit& dBPicoVolt();
protected: // instance members
    CPhysUnit m_physUnitPicoVolt;
    ..
    CPhysUnit m_physUnitVolt;
    CPhysUnit m_physUnitKiloVolt;
    CPhysUnit m_physUnitMegaVolt;
    CPhysUnit m_physUnitdBVolt;
    CPhysUnit m_physUnitdBu;
    ..
    CPhysUnit m_physUnitdBPicoVolt;

}; // class CPhysSizeVoltage
```

Im Konstruktor der Klasse *CPhysSizeVoltage* ist die Basisklasse *CPhysSize* sowie die physikalischen Einheiten zu parametrisieren:

```
//-------------------------------------------------------------------
CPhysSizeVoltage::CPhysSizeVoltage() :
//-------------------------------------------------------------------
    CPhysSize(
        /* strGroupName     */ "Electricity",
        /* strName          */ "Voltage",
        /* strSIUnitName    */ "Volt",
        /* strSIUnitSymbol  */ "V",
        /* strFormulaSymbol */ "U",
        /* bIsPowerRelated  */ false ),
```

Die Übergabeparameter an den Konstruktor der Klasse *CPhysSize* sind – nach Studium er Grundlagen - selbsterklärend, ebenso die Konstruktoren, um die Einheiten „PicoVolt“, „Volt“, „KiloVolt“ und „MegaVolt“ anzulegen.

```
    m_physUnitPicoVolt(
        /* physSize  */ *this,
        /* strPrefix */ "p" ),
    ..
    m_physUnitVolt(
        /* physSize  */ *this,
        /* strPrefix */ "" ),
    ..
    m_physUnitMegaVolt(
        /* physSize  */ *this,
        /* strPrefix */ "M" ),
```

Die Konstruktoren für die logarithmische Einheiten sind etwas komplexer, da hier der Name der Einheit und die Umrechnungsfaktoren nicht automatisch aus dem Präfix ermittelt werden können. Deshalb sind Name und Symbol der Einheit explizit anzugeben. Ferner wird der Referenzwert für die Logarithmus-Funktion benötigt. Übrigens wird der Konstruktor mit derselben Signatur auch dazu verwendet, um Einheiten anzulegen, die nicht über eine Potenzfunktion von der SI-Einheit abgeleitet werden können (wie z.B. „Inch“ in der Größenart „Länge“ oder „Grad“ der Größenart „Winkel“). Hierfür ist lediglich der Parameter „IsLogarithmic“ auf „false“ zu setzen.

```
    m_physUnitdBVolt(
        /* physSize       */ *this,
        /* bIsLogarithmic */ true,
        /* strName        */ "dBVolt",
        /* strSymbol      */ "dBV",
        /* fRefVal        */ 1.0 ),
    m_physUnitdBu(
        /* physSize       */ *this,
        /* bIsLogarithmic */ true,
        /* strName        */ "dB(0.775V)",
        /* strSymbol      */ "dBu",
        /* fRefVal        */ c_fRefValdBu_0775V ),
    m_physUnitdBPicoVolt(
        /* physSize       */ *this,
        /* bIsLogarithmic */ true,
        /* strName        */ "dBPicoVolt",
        /* strSymbol      */ "dBpV",
        /* fRefVal        */ c_fFactorPico )
```

Innerhalb des Konstruktors ist die Initialisierungs-Routine der Basis-Klasse CPhysSize aufzurufen. Diese ermittelt die SI-Einheit der Größenart und erzeugt die interne Umrechnungstabelle, mit der Einheiten innerhalb dieser Größenart umgerechnet werden können. Wird für den Parameter „*CreateFindBestChainedList*“ der Wert „*true*“ übergeben, werden die Einheiten automatisch für die „*findBestUnit*“ Funktionalität verkettet. Voraussetzung hierfür ist jedoch, dass die Einheiten beginnend mit der „kleinsten“ Einheit in aufsteigender Reihenfolge instanziiert werden, wie im Beispiel für die Größenart „Elektrische Spannung“ geschehen. Bei Erreichen der ersten logarithmischen Einheiten wird die Kette abgebrochen. „dBV“ gehört so nicht mehr zur Kette, die nach der bestmöglichen Einheit durchsucht wird.

```
// Call function of base class CPhysSize to initialize the
// physical size together with its units (e.g. to create the
// field with internal conversion routines and to create the
// chained list of Lower/Higher units).
initialize(true);
```

Innerhalb des Main-Modules “*ZSPhysSizesElectricityDllMain*” der Dll werden die physikalischen Größenarten erzeugt und Methoden zum exportieren der Größenarten bereitgestellt.

```
//---------------------------------------------------------------------
void ZS::PhysVal::Electricity::createPhysSizes()
//---------------------------------------------------------------------
{
    if( s_pPhysSizeCurrent == NULL )
    {
        s_pPhysSizeCurrent = new CPhysSizeCurrent();
    }
    if( s_pPhysSizeVoltage == NULL )
    {
        s_pPhysSizeVoltage = new CPhysSizeVoltage();
    }
    if( s_pPhysSizeResistance == NULL )
    {
        s_pPhysSizeResistance = new CPhysSizeResistance();
    }
    if( s_pPhysSizePower == NULL )
    {
        s_pPhysSizePower = new CPhysSizePower();
    }

} // createPhysSizes

CPhysSizeCurrent& Current();
CPhysSizeVoltage& Voltage();
CPhysSizeResistance& Resistance();
CPhysSizePower& Power();
```

Da für die Einheiten bei der Definition der physikalischen Größenart bereits Methoden definiert wurden, kann auf die Einheit Millivolt der elektrischen Größenart Spannung auf folgende Art zugegriffen werden:

```
CPhysUnit& mV = ZS::PhysVal::Electricity::Voltage().MilliVolt().
```

3.2. Umrechnungsfunktionen zwischen verschiedener Größenarten

Um Einheiten zwischen verschiedenen Größenarten umrechnen zu können, muss die Umrechnungsfunktion der physikalischen Größe zugewiesen werden. Ferner muss zuvor für die physikalische Größenart, die den Referenzwert für die Umrechnung zur Verfügung stellen muss, die virtuelle Methode *getRefVal* überschrieben werden. In unserem Beispiel wollen wir zwischen elektrischer Spannung, elektrischer Leistung und elektrischem Strom umrechnen. Die von *CPhysSize* abgeleitete Klasse CPhysSizeResistance liefert den Referenzwert 50 Ω.

```
//-------------------------------------------------------------------
double CPhysSizeResistance::getRefVal( CPhysUnit* i_pPhysUnitRef ) const
//-------------------------------------------------------------------
{
    double fValRef = 50.0; // Ohm

    if( i_pPhysUnitRef != NULL && i_pPhysUnitRef != &m_physUnitOhm )
    {
        fValRef = m_physUnitOhm.convertValue(fValRef,*i_pPhysUnitRef);
    }
    return fValRef;
}
```

Um die Umrechnungsfunktionen den physikalischen Größenwerten zuzuweisen, müssen diese zuvor mit ihren jeweiligen Einheiten erzeugt worden sein. Danach kann für jede Umrechnung die entsprechende Formel zugewiesen werden:

```
Voltage().addFctConvert( // P = U²/R
    /* physSizeDst    */ Power(),
    /* pPhysSizeRef   */ Resistance(),
    /* fctConvert     */ EFctConvert_SQRxDIVr );
Voltage().addFctConvert( // I = U/R
    /* physSizeDst    */ Current(),
    /* pPhysSizeRef   */ Resistance(),
    /* fctConvert     */ EFctConvert_xDIVr );
Power().addFctConvert( // U = sqrt(P*R)
    /* physSizeDst    */ Voltage(),
    /* pPhysSizeRef   */ Resistance(),
    /* fctConvert     */ EFctConvert_SQRT_xMULr_ );
Power().addFctConvert( // I = sqrt(P/R)
    /* physSizeDst    */ Current(),
    /* pPhysSizeRef   */ Resistance(),
    /* fctConvert     */ EFctConvert_SQRT_xDIVr_ );
Current().addFctConvert( // P = I²*R
    /* physSizeDst    */ Power(),
    /* pPhysSizeRef   */ Resistance(),
    /* fctConvert     */ EFctConvert_SQRxDIVr );
Current().addFctConvert( // U = I*R
    /* physSizeDst    */ Voltage(),
    /* pPhysSizeRef   */ Resistance(),
    /* fctConvert     */ EFctConvert_xMULr );
```

3.3. Konvertieren von Einheiten

Um einen Wert in eine vorgegebene Zieleinheit zu konvertieren, bietet die Klasse *CPhysUnit* zwei *convertValue* Methoden an. Mit der ersten Methode lässt sich ein Wert ohne Angabe eines Referenzwertes konvertieren. Wird dabei eine Konvertierung zwischen zwei verschiedenen Größenarten vorgenommen, holt sich die Methode den Default-Referenzwert von der physikalischen Größenart, die beim Festlegen der Umrechnungsfunktion über *CPhysSize::addFctConvert* angegeben wurde.

```
double fVal_mV = 3.0e9;
double fVal_600Ohm = 600.0;
double fVal_dBV;
double fVal_dBmW;
double fVal_BestVoltageUnit;
```

Bei nachfolgendem *convertValue* Aufruf wird der Wert innerhalb der Größenart **Voltage** konvertiert. Ein Referenzwert wird nicht benötigt.

```
fVal_dBV = Voltage().MilliVolt().convertValue(
    /* fVal        */ fVal_mV,
    /* physUnitDst */ Voltage().dBVolt() );
```

Bei nachfolgendem *convertValue* Aufruf wird der Wert von der Größenart **Voltage** in die Größenart **Power** konvertiert. Hierfür wird ein Referenzwiderstand **Ohm** benötigt, den sich die *convertValue* Methode von der physikalischen Größenart **Resistance** per *getValRef* Aufruf holt.

```
fVal_dBmW = Voltage().MilliVolt().convertValue(
    /* fVal        */ fVal_mV,
    /* physUnitDst */ Voltage().dBVolt() );
```

Bei nachfolgendem *convertValue* Aufruf wird der Wert von der Größenart **Voltage** in die Größenart **Power** konvertiert. Hierfür wird ein Referenzwiderstand **Ohm** benötigt, der über die zusätzlichen Parameter an die *convertValue* Methode übergeben wird und den die Konvertierungsfunktion zur Umrechnung verwendet.

```
fVal_dBmW = Voltage().MilliVolt().convertValue(
    /* fVal        */ fVal_mV,
    /* physUnitDst */ Power().dBMilliWatt(),
    /* fRefVal     */ fValRef_600Ohm,
    /* physUnitRef */ Resistance().Ohm() );
```

Um einen Wert in die bestmögliche Einheit innerhalb einer Größenart zu konvertieren, bietet die Klasse *CPhysUnit* die Methode *findBestUnit* an, die eine Referenz auf die gefundene, bestmögliche Einheit zurückgibt (diese kann die Ausgangseinheit selbst sein) sowie, falls gewünscht, den in die bestmögliche Einheit konvertierten Wert.

```
pPhysUnitBestVoltage = Voltage().MilliVolt().findBestUnit(
    /* fValSrc  */ fVal_mV,
    /* pfValRes */ &fVal_BestVoltageUnit );
```

3.4. Verwendung der Klasse *CPhysVal*

Die Klasse *CPhysVal* vereint Messwert und Einheit und kann eine Ungenauigkeit besitzen. Die Klasse bietet eine große Zahl von Konstruktoren an, um physikalische Werte mit und ohne Einheiten, mit und ohne Ungenauigkeiten, aus double Werten oder aus Strings zu erzeugen.

Nachfolgend wird ein physikalischer Wert über seinen Default-Konstruktor erzeugt. Der Wert ist noch ungültig. Erst über einen nachfolgenden setValue Aufruf erhält die PhysVal-Instanz einen gültigen Wert zugewiesen. Über die *toString* Methode kann der physikalische Wert in einen String konvertiert werden:

```
CPhysVal physVal;
physVal.setVal(45.0);
strPhysVal = physVal.toString();
```

Über nachfolgenden Konstruktur wird eine PhysVal-Instanz durch einen String erzeugt, der nicht nur den Wert, sondern auch Einheit und Fehlerwert beinhaltet. Auch hier wird über die *toString* Methode der physikalische Wert wieder in einen String konvertiert werden.

```
CPhysVal physVal( "1.234 Geometry.km ± 56 m", EResTypeAccuracy );
strActualValue = physVal.toString(
  /* unitFindVal          */ EUnitFindNone,
  /* iValSubStrVisibility */ EPhysValSubStrVal|EPhysValSubStrUnitSymbol,
  /* unitFindRes          */ EUnitFindNone,
  /* iResSubStrVisibility */ EPhysValSubStrVal );
```

Weitere, ausführlichere Beispiele können dem Modultest der Klasse *CPhysVal* entnommen werden, der sich innerhalb des Moduls „*ZSTestPhysVal.cpp*“ der Test-Applikation „*ZSAppPhysVal*“ befindet.

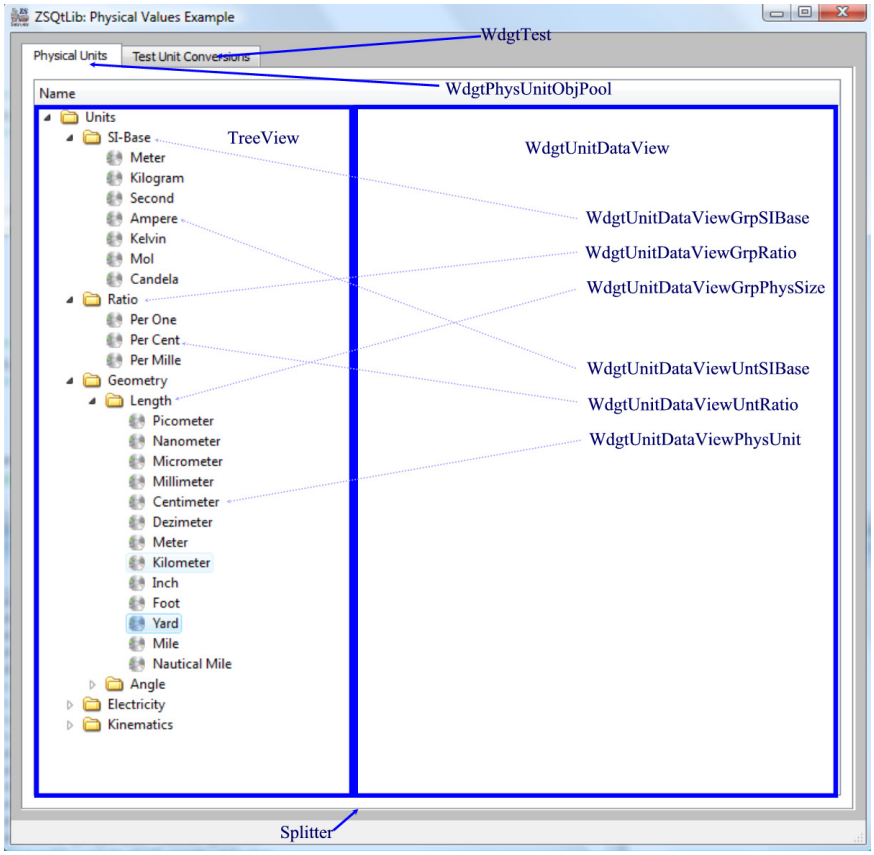
4. Beispiel-Applikation

Die Beispielapplikation besteht aus zwei Karteikarten:

1. Eine erste Karteikarte, um die Konfiguration der Größenarten und Einheiten visuell zu prüfen.
2. Eine zweite Karteikarte, um die Umrechnung zwischen den Einheiten manuell zu testen sowie einen automatisierten Test zu starten, der einen Klassentest von *CPhysVal* darstellt.

4.1. Prüfen der Konfiguration

In der ersten Karteikarte können die Attribute der konfigurierten Größenarten mit den jeweiligen Einheiten in übersichtlicher Form betrachtet werden. Dabei kann in der linken Seite der Karteikarte über eine Baumstruktur durch die konfigurierten Größenarten und Einheiten „gebrowst“ werden. Je nach selektiertem Eintrag innerhalb der Baumstruktur wird das „DataView“-Widget auf der rechten Seite der Karteikarte ausgetauscht, um den Inhalt des selektierten Eintrags wiederzugeben.



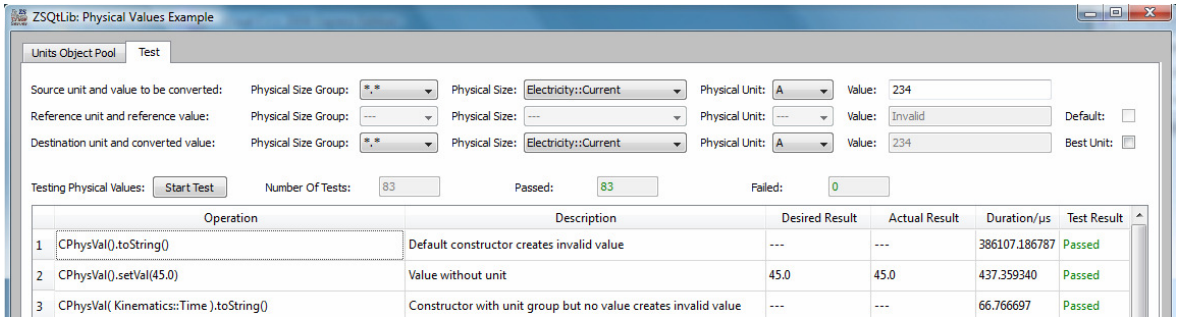
4.2. Test der Konvertierungsroutinen und Test der Klasse *CPhysVal*

Über die zweite Karteikarte können die Einheiten-Umrechnungen sowie die Klasse *CPhysVal* getestet werden.

Zum Test der Einheiten-Umrechnungen bietet die Karteikarte drei Zeilen mit Comboboxen und Eingabefeldern an.

1. In der ersten Zeile wird die umzurechnende Einheit über Combo-Boxen ausgewählt und über ein Textfeld der zu konvertierende Wert eingegeben.
2. In der zweiten Zeile kann ein Referenzwert eingegeben werden, falls zwischen zwei unterschiedlichen Größenarten umzurechnen ist. Die Combo-Boxen zur Auswahl des Wissensfeldes sowie der physikalischen Größe werden für den Referenzwert in Abhängigkeit von der gewählten Quell- und Zieleinheit automatisch gefüllt und können nicht editiert werden. Die Combo-Box zur Auswahl der Einheit als auch das Textfeld zur Eingabe des Referenzwertes können editiert werden. Bei aktivieren der „Default“ Check-Box wird die SI-Einheit der Referenzgröße in die Combo-Box sowie der Default-Referenzwert in das Textfeld übernommen. Immer bei Ändern des Referenzwerts wird geprüft, ob der aktuelle Wert dem Default-Referenzwert entspricht und die Check-Box entsprechend aktualisiert.
3. In der dritten Zeile kann die Zieleinheit ausgewählt werden, in die der Wert aus Zeile 1 umzurechnen ist. Wird zusätzlich noch die Check-Box „Best Unit“ angekreuzt, wird der Wert in der bestmöglichen Einheit dargestellt. In diesem Fall ist die Combo-Box zur Auswahl der Zieleinheit nicht editierbar und zeigt automatisch die bestmögliche Einheit an. Anzumerken wäre vielleicht an dieser Stelle, dass es nur für lineare Einheiten möglich ist, eine bestmögliche Einheit zur Darstellung des Wertes zu finden. Für logarithmische Einheiten kann die „Best Unit“ Check-Box zwar aktiviert werden, die aber zuvor selektierte, logarithmische Einheit sollte sich jedoch nicht ändern.

Zum Test der Klasse *CPhysVal* bietet die Karteikarte einen „Start“ Button an, Text-Felder zur Ausgabe der Anzahl der Einzeltests und des Testergebnisses, sowie eine Tabelle, in der die durchgeführten Einzeltests protokolliert werden. Der Test wurde so konzipiert, dass jeder Konstruktur und jede Methode mindestens einmal durchlaufen wird. Bei jeder Änderung innerhalb der Klassenbibliothek sollte der Test fehlerfrei durchgeführt werden. Werden Bugs behoben, sollte der Test so erweitert werden, dass dieser Bug reproduziert wird. Danach sollte der Bug behoben und der Test von neuem gestartet werden.



5. Methoden und Klassen-Referenz

5.1. Type Definitionen und Globale Methoden

Das Subsystem ZSPhysVal stellt einige globale Methoden zur Verfügung, die innerhalb des Pakets häufiger verwendete Code-Fragmente kapseln bzw. die es „verdient haben“, als Hilfsmethoden exportiert zu werden. In erster Linie handelt es sich dabei um Methoden, um Zahlenwerte in Strings zu konvertieren.

5.1.1. *Datentype (enum) EFormatResult*

Wird als Rückgabewert von Konvertierungsroutinen verwendet. Dabei wird das höherwertigste Bit (0x80) verwendet, um anzuzeigen, dass ein „echter“ Fehler aufgetreten ist und der Wert nicht konvertiert werden konnte. Ist dieses Bit nicht gesetzt, wurde entweder der Wert erfolgreich konvertiert (Ok) oder kann nicht ganz so wie gewünscht wiedergegeben werden. Um zu prüfen, ob ein „echter“ Fehler aufgetreten ist, kann der enum Wert *EFormatResultError* wie folgt verwendet werden:

```
if( formatResult & EFormatResultError )
{
    send error message “Value not convertible”.
}
```

Folgende enum Werte wurden definiert:

EFormatResultOk (0x00)
: Der Wert wurde erfolgreich konvertiert.

EFormatResultAccuracyOverflow (0x01) (Warning)
: Der Wert wird zu genau wiedergegeben. Z.B. bei der Darstellung ohne Exponent und ohne „findBestUnit“ muss der Wert „12345678 V“, der eine Ungenauigkeit von „1000 V“ besitzt, mit „12345700 V“ wiedergegeben werden, obwohl die beiden „0“len eine höhere Genauigkeit als 3 Volt anzeigen.

EFormatResultAccuracUnderflow (0x02) (Warning)
: Der Wert wird zu ungenau wiedergegeben. Z.B. bei der Darstellung mit einer vorgegebenen, maximalen Anzahl von sechs Ziffern für die Mantisse muss der Wert „12345,678 V“, der eine Ungenauigkeit von „0,1 V“ besitzt, mit „12345,7 V“ wiedergegeben werden, obwohl aufgrund der Ungenauigkeit zwei Nachkommastellen mit ausgegeben hätten werden sollen.

EFormatResultUnitConversionFailed (0x83) (Error)
: Der Wert konnte nicht konvertiert werden, weil ein Fehler bei der Konvertierung von Einheiten auftrat.

EFormatResultValueOverflow (0x01) (Error)
: Der Wert kann nicht wiedergegeben werden, da die vorgegeben Anzahl von Ziffern für die Mantisse und oder den Exponenten nicht ausreicht. Z.B. kann der Wert „12345678 V“ nicht ohne Exponenten und einer maximalen Anzahl von fünf Ziffern für die Mantisse ausgegeben werden.

EFormatResultValueUnderflow (0x01) (Error)
: Der Wert kann nicht wiedergegeben werden, da die vorgegeben Anzahl von Ziffern für die Mantisse und oder den Exponenten nicht ausreicht. Z.B. kann der Wert „0,0000012345678 V“ nicht ohne Exponenten und einer maximalen Anzahl von fünf

Ziffern für die Mantisse ausgegeben werden. Eigentlich nur ein „halber Fehler“, da der Wert dennoch mit „0“ ausgegeben werden könnte.

5.1.2. *Datentype (enum) EResType*

Legt fest, ob es sich bei einer Ungenauigkeitsangabe eines physikalischen Wertes um die Angabe der Genauigkeit oder um die Auflösung des Wertes handelt. Die Anzahl der im Ergebnisstring ausgegebenen, „unsicheren“ Dezimalstellen wird anhand dieses Types wie folgt festgelegt.

EResTypeAccuracy
: Es werden zwei unsichere Stellen ausgegeben, falls die erste Ziffer der Ungenauigkeit eine 1 oder 2 ist, ansonsten wird nur eine unsichere Stelle ausgegeben.

EResTypeResolution
: Es wird immer nur eine unsichere Stelle ausgegeben und der Wert wird (zur Ausgabe) auf ein ganzzahliges Vielfaches der Resolution gerundet.

5.1.3. *Datentype (enum) EUnitFind*

Legt fest, ob ein mit einer Enheit behafteter Wert in seiner bestmöglichen Einheit ausgegeben werden soll.

EUnitFindNone
: Wert wird in der vorgegebenen Einheit ausgegeben.

EResTypeResolution
: Wert wird in seiner „bestmöglichen“ Einheit ausgegeben.

5.1.4. *Datentype (enum) EPhysValSubStr*

Ein String, der die Angabe eines physikalischen Wertes enthält, wird in folgende Teilstrings aufgeteilt:

„12,345 Kinematics::Time::ms“

EPhysValSubStrNone (0x00)
: Kein Teilstring.

EPhysValSubStrVal (0x01)
: Kennzeichnet den Wert („12,345“)

EPhysValSubStrUnitMainGroup (0x02)
: Kennzeichnet die Angabe des Wissensgebiets („Kinematics“)

EPhysValSubStrUnitSubGroup (0x04)
: Kennzeichnet die Angabe der Größenart („Time“)

EPhysValSubStrUnitPrefix (0x08)
: Kennzeichnet die Angabe des Präfixes des Einheitensymbols („m“).

EPhysValSubStrUnitSymbol (0x10)
: Kennzeichnet die Angabe des Einheitensymbols („ms“).

EPhysValSubStrUnitName (0x10)
: Kennzeichnet die Angabe des Einheitennamens („MilliSecond“).

Bei der Konvertierung physikalischer Werte oder Ungenauikgeitsangaben in Strings kann durch bitweise Veroderung über diese Enum-Werte festgelegt werden, welche Teilstrings erzeugt werden sollen. Die Bits UnitPrefix, UnitName und UnitSymbol schliessen sich jedoch gegenseitig aus.

5.1.5. *Datentype (enum) EValidity*

Für physikalische Werte muss berücksichtigt werden, dass sie

- ungültig sein können (wenn die Messung gerade gestartet wurde und der Messwert noch nicht vorliegt),
- außerhalb des darstellbaren Zahlbereichs liegen können,
- zu genau wiedergegeben werden müssen, weil in einer Darstellung ohne Exponenten Nullen vor dem Komman eingefügt werden müssen, um den Wert darzustellen,
- zu ungenau wiedergegeben werden müssen, weil nicht genügend Platz zur Verfügung steht, um alle gültigen Stellen darzustellen,
- als Zwischenwerte vorliegen, weil eine Messung sehr lange dauert, und man Zwischenergebnisse darstellen will.

Diese Zustände werden über ein Validity angezeigt, das eine bitwise Veroderung folgender enum Werte zusammengesetzt ist:

EInvalid (0x00)
: Wert ist noch nicht gültig (wurde noch nicht initialisiert).

EValid (0x01)
: Wert ist gültig.

Weitere Flags wurden bislang noch nicht benötigt und müssten im Bedarfsfall hinzugefügt werden.

5.1.6. *Datentype (struct) SValueFormatProvider*

Fasst Formatierungsanweisungen, mit denen ein physikalischer Wert in einen String konvertiert werden kann, in einer Struktur zusammen. Die Struktur hat folgende, „öffentliche“ Attribute:

m_pUnitVal (IN) Datentype: CUnit*
: Einheit des zu formatierenden Wertes. Darf auch NULL sein, wenn der Wert keine Einheit besitzt.

m_unitFindVal Datentype: *EUnitFind*
: Definiert, ob der Wert in seine bestmögliche Einheit zu konvertieren ist.

m_iValSubStrVisiblility (IN) Datentype: int
: Verodertes Bit-Feld über das festgelegt wird, welche Teilstrings für den Größenwert auszugeben sind (siehe enum EPhysValSubStr).

m_iDigitsMantissa (IN) Datentype: int
: Ein Wert größer als 0 begrenzt die maximale Anzahl der Ziffern für die Mantisse (Vorkomma- und Nachkommastring). Wird benützt, um den resultierenden String auf eine maximale Zahl von Zeichen zu begrenzen.

m_iDigitsExponent (IN) Datentype: int
Ein Wert größer oder gleich 0 begrenzt die maximale Anzahl der Ziffern für den Exponenten (für 0 wird der Wert ohne Exponent ausgegeben).

m_bUseEngineeringFormat (IN) Datentype: bool (Default = false)
Legt fest, ob der Wert so formatiert werden soll, dass er mit genau einer Vorkommastelle ausgegeben wird. Für diesen Fall ist ggf. (je nach Wert und Einheit) ein Exponent notwendig.

m_iDigitsPerDigitGroup (IN) Datentype: int
Ein Wert größer als 0 bestimmt die Anzahl der Ziffern pro „Tausender-Gruppe". Falls *DigitsPerDigitGroup* ungleich 3 ist, ist „Tausender-Gruppe" natürlich der falsche Ausdruck, aber verdeutlicht doch den Sachverhalt. Als Trennzeichen wird der innerhalb *DigitsGroupDelimiter* referenzierte String verwendet. Wird ein NULL Pointer für den String übergeben, werden Leerzeichen zur Trennung der Ziffern-Gruppen verwendet. Beispiel:

```
1. Value = 1234567.8901234
   DigitsPerDigitGroup = 3
   DigitsGroupDelimiter = NULL
   Ergebnis: "1 234 567.890 123 4"
2. Value = 1234567.8901234
   DigitsPerDigitGroup = 4
   DigitsGroupDelimiter = "#"
   Ergebnis: "123#4567.8901#234"
```

m_pstrDigitsPerDigitGroup (IN) Datentype: QString*
Siehe *DigitsPerDigitGroup*.

m_pstrDecimalPoint (IN) Datentype: QString*
Als Default wird ein Punkt als Dezimalkomma verwendet. Um statt diesem Punkt ein anderes Zeichen (z.B. wie im Deutschen üblich ein Komma) zu verwenden, muss hier ein gültiger Verweis auf den String übergeben werden, der das zu verwendende Dezimalzeichen enthält.

m_fRes (IN) Datentype: double
Größenwert der Ungenauigkeitsangabe (ein Wert = 0.0 legt fest, dass der Wert keine Ungenauigkeit besitzt).

m_pUnitRes (IN) Datentype: CUnit*
Einheit der Ungenauigkeitsangabe. Darf auch NULL sein, wenn der Wert entweder keine Einheit besitzt oder wenn die Ungenauigkeit in derselben Einheit vorliegt, wie der physikalische Wert.

i_resType (IN) Datentype: EResType
Legt fest, ob es sich bei *fRes* um eine Ungenauigkeit oder um die Auflösung des zu formatierenden Wertes handelt. Die Anzahl der im Ergebnisstring ausgegebenen, „unsicheren" Dezimalstellen wird anhand des „ResType" festgelegt.
Für „*resType = EResTypeAccuracy*" werden zwei unsichere Stellen ausgegeben, falls die erste Ziffer der Ungenauigkeit eine 1 oder 2 ist, ansonsten wird nur eine unsichere Stelle ausgegeben.
Für „*resType = EResTypeResolution*" wird immer nur eine unsichere Stelle ausgegeben und der Wert wird (zur Ausgabe) auf ein ganzzahliges Vielfaches der Resolution gerundet.

m_unitFindRes Datentype: *EUnitFind*
Definiert, ob die Ungenauigkeit in die bestmögliche Einheit zu konvertieren ist.

m_iResSubStrVisiblility (IN) Datentype: int
Verodertes Bit-Feld über das festgelegt wird, welche Teilstrings für die Ungenauigkeitsangabe auszugeben sind (siehe enum EPhysValSubStr).

m_iDigitsAccuracy (IN) Datentype: int
Ein Wert größer als 0 begrenzt entweder die Anzahl der zu verwendenen Nachkommasellen oder die Anzahl von Ziffern für die gesamte Mantisse. Im Gegensatz zu *DigitsMantissa* kann der resultierende String trotzdem mehr Ziffern besitzen, als durch *DigitsAccuracy* vorgegeben wurden. Das Ergebnis der Formatierung ist dann ein *AccuracyOverflow*. Zu beachten ist, dass führende Nullen nicht für die Genauigkeit relevant sind. Soll z.B. ein Wert mit einer Genauigkeit von fünf Stellen angezeigt werden, wird der Wert 0.0003456789 in den String 0.00034567 konvertiert (sofern dabei nicht *DigitsMantissa* überschritten wird).
Die Angabe von DigitsAccuracy > 0 und einer Ungenauigkeitsangabe von fRes > 0.0 schliesst sich gegenseitig aus, wird aber nicht unbedingt als Fehler behandelt. Innerhalb der Klasse CPhysVal besitzt *fRes* gegenüber *DigitsAccuracy* Vorrang.
i_bDigitsAccuracyLimitMantissa (IN) Datentype: bool
Legt fest, ob über *DigitsAccuracy* die Anzahl von Ziffern der Nachkommastellen oder die Anzahl der Ziffern für die gesamte Mantisse begrenzt werden soll.

Die Struktur besitzt ferner eine Default-Konstruktor, einen Copy-Konsruktor sowie Konstrukturen, über die beim Anlegen der Struktur eine sinnvolle Kombination der Attribute festgelegt werden kann.

5.1.7. *getPrefixStrFromFactor(double, QString*)*

Beschreibung:
Gibt für den übergebenen Faktor den zugehörigen Prefix-String zurück (z.B. „m“ für 10^{-3}, „k“ für 10^{3}). Ist der Faktor nicht dekadisch, wird ein „?“ zurückgegeben.
Parameter:
i_fFactor (IN) Datentype: double
Faktor.
o_pStrPrefix (IN) Datentype: QString*
Hier wird der Prefix-String zurückgegeben.
Rückgabewert: .....Datentype: bool

5.1.8. *getExponentFromFactor(double, int*)*

Beschreibung:
Gibt für den übergebenen, dekadischen Faktor den zugehörigen Exponenten zurück (z.B. -3 für 10^{-3}).
Parameter:
i_fFactor (IN) Datentype: double
Faktor.
o_piExponent (IN) Datentype: int*
Hier wird der ermittelte Exponent zurückgegeben.
Rückgabewert: Datentype: bool
Konnte der Exponent nicht ermittelt werden, weil der Faktor nicht dekadisch ist, wird false zurückgegeben.

5.1.9. *getFactorInverted(double, double*)*

Beschreibung:
Gibt für den übergebenen, dekadischen Faktor den zugehörigen, „inversen“ Faktro zurück (z.B. 10^{3} für 10^{-3}).
Parameter:
i_fFactor (IN) Datentype: double
Faktor.
o_pfFactorInverted (IN) Datentype: double*
Hier wird der ermittelte, inverse Faktor zurückgegeben.
Rückgabewert: Datentype: bool
Konnte der inverse Faktor nicht ermittelt werden, weil der Faktor nicht dekadisch ist, wird false zurückgegeben.

5.1.10. *getFactorFromPrefixStr(const QString&, bool)*

Beschreibung:
Gibt für den übergebenen Präfix-String den zugehörigen, dekadischen Faktor zurück (z.B. „m“ für 10^{-3}).
Parameter:
i_strPrefix (IN) Datentype: QString
Präfix-String.
i_bInverted (IN) Datentype: bool
true, falls der inverse Faktor erwünscht ist.
Rückgabewert: Datentype: double
Konnte der Faktor nicht ermittelt werden, weil der Präfix-String ungültig war, wird der „neutrale“ Faktor 1.0 zurückgegeben.

5.1.11. *getExponentFromPrefixStr(const QString&)*

Beschreibung:
Gibt für den übergebenen Präfix-String den zugehörigen Exponenten zurück (z.B. „6“ für „G“).
Parameter:
i_strPrefix (IN) Datentype: QString
Präfix-String.
Rückgabewert: Datentype: int
Konnte der Exponent nicht ermittelt werden, weil der Präfix-String ungültig war, wird der „neutrale“ Exponent 0 zurückgegeben.

5.1.12. *getExponentStrFromPrefixStr(const QString&)*

Beschreibung:
Gibt für den übergebenen Präfix-String einen Exponenten-String zurück (z.B. „Giga“ für „G“). Eigentlich ist der Name der Methode nicht glücklich gewählt, aber sie wird nur für Debugging- und Test-Zwecke benötigt.
Parameter:
i_strPrefix (IN) Datentype: QString
Präfix-String.

Rückgabewert: Datentype: QString
Konnte der Exponent nicht ermittelt werden, weil der Präfix-String ungültig war, wird ein „?“ zurückgegeben.

5.1.13. *areOfSameUnitGroup(const CUnit*, const CUnit*)*

Beschreibung:
Prüft, ob die übergebenen Einheiten zur selben Größenart gehören. Dabei berücksichtigt die Methode, dass die Eingabeparameter NULL Pointer sein können. Sind beide übergebenen Einheiten NULL Pointer, betrachtet die Methode die Einheiten als zur selben Größenart (keine Größenart, also einheitenlos) gehörig. Ist nur eine der beiden Einheiten ein NULL Pointer, gehören die beiden Einheiten nicht zur selben Größenart.
Parameter:
i_pUnit1 (IN) Datentype: const CUnit*
i_pUnit2 (IN) Datentype: const CUnit*
Rückgabewert: Datentype: bool

5.1.14. *getSymbol(CUnit*)*

Beschreibung:
Methode, die zur Vereinfachung (Verkürzung) des Codes verwendet werden kann. Sie prüft, ob die übergebene Einheit ein gültiger Zeiger ist. Wenn ja, wird das Symbol dieser Einheit zurückgegeben, wenn nicht, wird ein Leerstring zurückgegeben.
Parameter:
i_pUnit (IN) Datentype: const CUnit*
Rückgabewert: Datentype: QString

5.1.15. *getName(CUnit*)*

Beschreibung:
Methode, die zur Vereinfachung (Verkürzung) des Codes verwendet werden kann. Sie prüft, ob die übergebene Einheit ein gültiger Zeiger ist. Wenn ja, wird der Name dieser Einheit zurückgegeben, wenn nicht, wird ein Leerstring zurückgegeben.
Parameter:
i_pUnit (IN) Datentype: const CUnit*
Rückgabewert: Datentype: QString

5.1.16. *insertDelimiter(int, const QString, QString*, int, int)*

Beschreibung:
Fügt „Tausendertrennzeichen“ in einen String ein.
Parameter:
i_iDigitsPerDigitGroup (IN) Datentype: int
Anzahl der Ziffern die zu einer „Tausender-Gruppe“ zusammengefasst werden sollen.
i_strDelimiter (IN) Datentype: QString
Das Tausendertrennzeichen, das eingefügt werden soll.

o_pStrValue (IN/OUT) Datentype: QString*
String, in den die Tausendertrennzeichen einzufügen sind.
i_iDigitsLeading (IN) Datentype: int
Anzahl der Ziffern innerhalb *StrValue* vor dem Dezimalpunkt.
i_iDigitsTrailing (IN) Datentype: int
Anzahl der Ziffern innerhalb *StrValue* nach dem Dezimalpunkt.

5.1.17. *removeTrailingZeros(QString*, unsigned int, QChar)*

Beschreibung:
Entfernt am Ende vorkommende „0“len aus dem Nachkommastring.
Parameter:
io_pstrValue (IN/OUT) Datentype: QString*
String, aus dem die „0“ entfernt werden sollen.
i_uDigitsTrailingMin (IN) Datentype: int
Anzahl der Nachkommastellen, die beibehalten werden sollen (Default = 1)
i_charDecPoint (IN) Datentype: QChar
Dezimal-Trennzeichen (Default = ‚.’).
Rückgabewert: Datentype: EFormatResult

5.1.18. *formatString(const QString&, int*, int*, int*)*

Beschreibung:
Untersucht den Eingabestring und gibt die Anzahl der Zeichen zurück, die vor dem Dezimalpunkt, nach dem Dezimalpunkt und innerhalb des Exponenten liegen (incl. Vorzeichen und Exponenten-Zeichen).
Parameter:
i_strValue Datentype: QString
Zu untersuchender Value-String.
i_iDigitsLeading (OUT) (Optional) Datentype: int*
Anzahl der Zeichen vor dem Dezimalpunkt.
i_iDigitsTrailing (OUT) (Optional) Datentype: int*
Anzahl der Zeichen nach dem Dezimalpunkt.
i_iDigitsExponent (OUT) (Optional) Datentype: int*
Anzahl der Zeichen im Exponenten.
Rückgabewert:.....Datentype: EFormatResult

5.1.19. *formatValue(double, CUnit*, double, CUnit*, EResType, int, int, bool, double*, QString*, CUnit**, int*, int*, int*)*

Beschreibung:
Formatiert den mit einer Einheit und einer Ungenauigkeit behafteten, eingegebenen Wert in einen String unter Berücksichtigung der festgelegten Formatierungsanweisungen und einer (möglichen) Konvertierung in die „bestmögliche“ Einheit zur Darstellung des Wertes.
Parameter:
i_fVal (IN) Datentype: double
Zu formatierender Zahlenwert.

i_pUnitVal (IN) Datentype: CUnit*
Einheit des zu formatierenden Wertes. Darf auch NULL sein, wenn der Wert keine Einheit besitzt.

i_fRes (IN) Datentype: double
Ungenauigkeit bzw. Auflösung des zu formatierender Zahlenwerts.

i_pUnitVal (IN) Datentype: CUnit*
Einheit der Ungenauigkeit bzw. der Auflösung des zu formatierenden Wertes. Darf auch NULL sein, wenn entweder der Wert keine Einheit besitzt oder für die Ungenauigkeit dieselbe Einheit verwendet werden soll, wie für den zu formatierenden Wert.

i_resType (IN) Datentype: EResType
Legt fest, ob es sich beim Parameter *fRes* um eine Ungenauigkeit oder um die Auflösung des zu formatierenden Wertes handelt. Die Anzahl der im Ergebnisstring ausgegebenen, „unsicheren" Dezimalstellen wird anhand des „ResType" festgelegt.
Für „*resType = EResTypeAccuracy*" werden zwei unsichere Stellen ausgegeben, falls die erste Ziffer der Ungenauigkeit eine 1 oder 2 ist, ansonsten wird nur eine unsichere Stelle ausgegeben.
Für „*resType = EResTypeResolution*" wird immer nur eine unsichere Stelle ausgegeben und der Wert wird (zur Ausgabe) auf ein ganzzahliges Vielfaches der Resolution gerundet.

i_iDigitsMantissa (IN) Datentype: int
Ein Wert größer als 0 begrenzt die maximale Anzahl der Ziffern für die Mantisse (Vorkomma- und Nachkommastring). Wird benützt, um den resultierenden String auf eine maximale Zahl von Zeichen zu begrenzen.

i_iDigitsExponent (IN) Datentype: int
Ein Wert größer oder gleich 0 begrenzt die maximale Anzahl der Ziffern für den Exponenten (für 0 wird der Wert ohne Exponent ausgegeben).

i_bUseEngineeringFormat (IN) Datentype: bool (Default = false)
Legt fest, ob der Wert so formatiert werden soll, dass er mit genau einer Vorkommastelle ausgegeben wird. Für diesen Fall ist ggf. (je nach Wert und Einheit) ein Exponent notwendig.

o_pfVal (OUT) Datentype: double* (Default = NULL)
Falls der Wert in der "bestmöglichen" Einheit dargestellt werden soll, wird der Wert konvertiert. Will man den die „bestmöglichen" Einheit konvertierten Wert wissen, muss man hier einen gültigen Zeiger übergeben.

o_pstrVal (OUT) Datentype: QString* (Default = NULL)
Soll der Wert in seine String-Darstellung konvertieren, ist hier ein gültiger Zeiger zu übergeben, in den der resultierende String abgelegt wird.

o_ppUnitVal (OUT) Datentype: CUnit** (Default = NULL)
Falls der Wert in der "bestmöglichen" Einheit dargestellt werden soll, muss hier ein gültiger Zeiger auf einen Einheiten-Zeiger übergeben werden. Z.B.

```
CUnit* pUnitBest = &Electricity::Voltage().Volt().
formatValue( … &pUnitBest ).
```

Nach Aufruf der Methode verweist *pUnitBest* auf die ermittelte, bestmögliche Einheit, um den Wert darzustellen.

o_piDigitsLeading (OUT) Datentype: int* (Default = NULL)
Wird hier ein gültiger Zeiger übergeben, enthält dieser Parameter nach Aufruf der Methode die Anzahl der Vorkommastellen.

o_piDigitsTrailing (OUT) Datentype: int* (Default = NULL)
Wird hier ein gültiger Zeiger übergeben, enthält dieser Parameter nach Aufruf der Methode die Anzahl der Nachkommastellen.
o_piDigitsExponent (OUT) Datentype: int* (Default = NULL)
Wird hier ein gültiger Zeiger übergeben, enthält dieser Parameter nach Aufruf der Methode die Anzahl der Ziffern des Exponenten. Das Exponenten-Zeichen ‚e' sowie das Vorzeichen ist immer im Exponenten-String enthalten, wird aber hier nicht zu den Exponent-Digits dazugezählt. Für den String „12.34e+21" ist *DigitsExponent* gleich 2.
Rückgabewert: .....Datentype: EFormatResult
Beispiel:

```
formatResult = formatValue(
    /* fVal                */ 123.45,
    /* pUnitVal            */ &Electricity::Power().MilliWatt(),
    /* fRes                */ 0.02,
    /* pUnitRes            */ NULL,
    /* resType             */ EResTypeResolution,
    /* iDigitsMantissa     */ 0,
    /* iDigitsExponent     */ 2,
    /* bUseEngineeringForm */ true,
    /* pfVal               */ &fValResult,
    /* pstrVal             */ &strValResult,
    /* ppUnitVal           */ &pUnitValResult,
    /* piDigitsLeading     */ &iDigitsLeadingResult,
    /* piDigitsTrailing    */ &iDigitsTrailingResult,
    /* piDigitsExponent    */ &iDigitsExponentResult );
```

Ergebnis(se):
fValResult = 123.46 (auf „*fRes*" gerundet)
strValResult = „1.2346e+2" (Engineering Format, auf „*fRes*" gerundet)
pUnitValResult = MilliWatt (entspricht der bestmöglichen Einheit)
iDigitsLeadingResult = 1 (eine Vorkommastelle)
iDigitsTrailingResult = 4 (vier Nachkommastellen)
iDigitsExponentResult = 1 (ein Digit im Exponenten)

5.1.20. *formatValue(double, CUnit*, double, CUnit*, EResType, int, int, int, const QString*, const QString*, bool, double*, QStirng*, CUnit**, int*, int*, int*)*

Beschreibung:
Formatiert den mit einer Einheit und einer Ungenauigkeit behafteten, eingegebenen Wert in einen String unter Berücksichtigung der festgelegten Formatierungsanweisungen und einer (möglichen) Konvertierung in die „bestmögliche" Einheit zur Darstellung des Wertes. Zusätzlich erlaubt diese Methode, Tausender-Trennzeichen zu verwenden und das Zeichen für den Dezimalpunkt anzugeben.
Parameter:
i_fVal (IN) Datentype: double
siehe obige *formatValue* Methode
i_pUnitVal (IN) Datentype: CUnit*
siehe obige *formatValue* Methode
i_fRes (IN) Datentype: double
siehe obige *formatValue* Methode

i_pUnitVal (IN) Datentype: CUnit*
siehe obige *formatValue* Methode
i_resType (IN) Datentype: EResType
siehe obige *formatValue* Methode
i_iDigitsMantissa (IN) Datentype: int
siehe obige *formatValue* Methode
i_iDigitsExponent (IN) Datentype: int
siehe obige *formatValue* Methode
i_iDigitsPerDigitGroup (IN) Datentype: int
Ein Wert größer als 0 bestimmt die Anzahl der Ziffern pro „Tausender-Gruppe“. Falls *DigitsPerDigitGroup* ungleich 3 ist, ist „Tausender-Gruppe“ natürlich der falsche Ausdruck, aber verdeutlicht doch den Sachverhalt. Als Trennzeichen wird der über den Parameter *DigitsGroupDelimiter* übergebene String verwendet. Wird ein NULL Pointer für den Sring übergeben, werden Leerzeichen zur Trennung der Ziffern-Gruppen verwendet. Beispiel:

```
3. Value = 1234567.8901234
   DigitsPerDigitGroup = 3
   DigitsGroupDelimiter = NULL
   Ergebnis: “1 234 567.890 123 4”
4. Value = 1234567.8901234
   DigitsPerDigitGroup = 4
   DigitsGroupDelimiter = “#”
   Ergebnis: “123#4567.8901#234”
```

i_pstrDigitsPerDigitGroup (IN) Datentype: QString*
Siehe Parameter *DigitsPerDigitGroup*.
i_pstrDecimalPoint (IN) Datentype: QString*
Als Default wird ein Punkt als Dezimalkomma verwendet. Um statt diesem Punkt ein anderes Zeichen (z.B. wie im Deutschen üblich ein Komma) zu verwenden, muss hier ein gültiger Verweis auf den String übergeben werden, der das zu verwendende Dezimalzeichen enthält.
i_bUseEngineeringFormat (IN) Datentype: bool (Default = false)
siehe obige *formatValue* Methode
o_pfVal (OUT) Datentype: double* (Default = NULL)
siehe obige *formatValue* Methode
o_pstrVal (OUT) Datentype: QString* (Default = NULL)
siehe obige *formatValue* Methode
o_ppUnitVal (OUT) Datentype: CUnit** (Default = NULL)
siehe obige *formatValue* Methode
o_piDigitsLeading (OUT) Datentype: int* (Default = NULL)
siehe obige *formatValue* Methode
o_piDigitsTrailing (OUT) Datentype: int* (Default = NULL)
siehe obige *formatValue* Methode
o_piDigitsExponent (OUT) Datentype: int* (Default = NULL)
siehe obige *formatValue* Methode
Rückgabewert: Datentype: EFormatResult

5.1.21. *formatValue(double, CUnit*, int, bool, int, int, bool, double*, QString*, CUnit**, int*, int*, int*)*

Beschreibung:

Formatiert den eingegebenen Wert in einen String unter Berücksichtigung der festgelegten Formatierungsanweisungen und einer (möglichen) Konvertierung in die „bestmögliche“ Einheit zur Darstellung des Wertes. Dabei wird die Genauigkeit nicht über einen mit einer Einheit behafteten Wert festgelegt, sondern über die Anzahl gültiger Ziffern. Diese Methode bietet sich also an, um einen Wert z.B. immer mit einer festgelegten Anzahl von Nachkommastellen auszugeben.

Parameter:

i_fVal (IN) Datentype: double
siehe obige *formatValue* Methode

i_pUnitVal (IN) Datentype: CUnit*
siehe obige *formatValue* Methode

i_iDigitsMantissaMax (IN) Datentype: int
Ein Wert größer als 0 begrenzt die maximale Anzahl der Ziffern für die Mantisse (Vorkomma- und Nachkommastring). Wird benützt, um den resultierenden String auf eine maximale Zahl von Zeichen zu begrenzen.

i_bDigitsAccuracyLimitMantissa (IN) Datentype: bool
Legt fest, ob der nachfolgende Eingabe-Parameter DigitsAccuracy die Anzahl von Ziffern der nachkommastellen oder die Anzahl der Ziffern für die gesamte Mantisse begrenzen soll.

i_iDigitsAccuracy (IN) Datentype: int
Ein Wert größer als 0 begrenzt entweder die Anzahl der zu verwendenen Nachkommasellen oder die Anzahl von Ziffern für die gesamte Mantisse. Im Gegensatz zum Eingabeparameter *DigitsMantissaMax* kann der resultierende String trotzdem mehr Ziffern besitzen, als durch *DigitsAccuracy* vorgegeben wurden. Das Ergebnis der Formatierung ist dann ein *AccuracyOverflow*. Zu beachten ist, dass führende Nullen nicht für die Genauigkeit relevant sind. Soll z.B. ein Wert mit einer Genauigkeit von fünf Stellen angezeigt werden, wird der Wert 0.0003456789 in den String 0.00034567 konvertiert (sofern dabei nicht *DigitsMantissaMax* überschritten wird).

i_iDigitsExponent (IN) Datentype: int
siehe obige *formatValue* Methode

i_bUseEngineeringFormat (IN) Datentype: bool (Default = false)
siehe obige *formatValue* Methode

o_pfVal (OUT) Datentype: double* (Default = NULL)
siehe obige *formatValue* Methode

o_pstrVal (OUT) Datentype: QString* (Default = NULL)
siehe obige *formatValue* Methode

o_ppUnitVal (OUT) Datentype: CUnit** (Default = NULL)
siehe obige *formatValue* Methode

o_piDigitsLeading (OUT) Datentype: int* (Default = NULL)
siehe obige *formatValue* Methode

o_piDigitsTrailing (OUT) Datentype: int* (Default = NULL)
siehe obige *formatValue* Methode

o_piDigitsExponent (OUT) Datentype: int* (Default = NULL)
siehe obige *formatValue* Methode

Rückgabewert: Datentype: EFormatResult

5.1.22. *formatValue(double, CUnit*, int, bool, int, int, int, const QString*, cont QString*, bool, double*, QString*, CUnit**, int*, int*, int*)*

Beschreibung:
Formatiert den eingegebenen Wert in einen String unter Berücksichtigung der festgelegten Formatierungsanweisungen und einer (möglichen) Konvertierung in die „bestmögliche“ Einheit zur Darstellung des Wertes. Dabei wird die Genauigkeit nicht über einen mit einer Einheit behafteten Wert festgelegt, sondern über die Anzahl gültiger Ziffern. Diese Methode bietet sich also an, um einen Wert z.B. immer mit einer festgelegten Anzahl von Nachkommastellen auszugeben. Zusätzlich erlaubt diese Methode, Tausender-Trennzeichen zu verwenden und das Zeichen für den Dezimalpunkt anzugeben.
Parameter:
i_fVal (IN) Datentype: double
siehe obige *formatValue* Methode
i_pUnitVal (IN) Datentype: CUnit*
siehe obige *formatValue* Methode
i_iDigitsMantissaMax (IN) Datentype: int
siehe obige *formatValue* Methode
i_bDigitsAccuracyLimitMantissa (IN) Datentype: bool
siehe obige *formatValue* Methode
i_iDigitsAccuracy (IN) Datentype: int
siehe obige *formatValue* Methode
i_iDigitsExponent (IN) Datentype: int
siehe obige *formatValue* Methode
i_iDigitsPerDigitGroup (IN) Datentype: int
siehe obige *formatValue* Methode
i_pstrDigitsPerDigitGroup (IN) Datentype: QString*
siehe obige *formatValue* Methode
i_pstrDecimalPoint (IN) Datentype: QString*
siehe obige *formatValue* Methode
i_bUseEngineeringFormat (IN) Datentype: bool (Default = false)
siehe obige *formatValue* Methode
o_pfVal (OUT) Datentype: double* (Default = NULL)
siehe obige *formatValue* Methode
o_pstrVal (OUT) Datentype: QString* (Default = NULL)
siehe obige *formatValue* Methode
o_ppUnitVal (OUT) Datentype: CUnit** (Default = NULL)
siehe obige *formatValue* Methode
o_piDigitsLeading (OUT) Datentype: int* (Default = NULL)
siehe obige *formatValue* Methode
o_piDigitsTrailing (OUT) Datentype: int* (Default = NULL)
siehe obige *formatValue* Methode
o_piDigitsExponent (OUT) Datentype: int* (Default = NULL)
siehe obige *formatValue* Methode
Rückgabewert: Datentype: EFormatResult

5.2. Klasse *CFctConvert*

Beschreibt die anzuwendenden Umrechnungsfunktionen, um einen Wert aus einer Quelleinheit in eine Zieleinheit umzurechnen und bietet einige Hilfsfunktionen an, um die Funktion in einem „lesbaren“ String darzustellen.

5.2.1. Attribute

m_pPhysUnitSrc Datentype: CPhysUnit*
: Referenz auf die Quell-Einheit.

m_pPhysUnitDst Datentype: CPhysUnit*
: Referenz auf die Quell-Einheit.

m_pPhysUnitRef Datentype: CPhysUnit*
: Referenz auf die Referenzeinheit, falls Quell- und Ziel-Einheit nicht derselben Größenart angehören.

m_fctConvertType Datentype: EFctConvertType
: Definiert die mathematische Funktion, die angewendet werden soll (siehe Erläuterung unten).

m_strFctConvertName Datentype: QString
: Mathematische Funktion in „lesbarer“ Form als String.

m_fM Datentype: double
: Wert, der für den Parameter „m“ in die Funktion eingesetzt wird.

m_fT Datentype: double
: Wert, der für den Parameter „t“ in die Funktion eingesetzt wird.

m_fK Datentype: double
: Wert, der für den Parameter „k“ in die Funktion eingesetzt wird.

Erläuterung:

Folgende mathematischen Funktionen sind über ein Enum definiert:

```
typedef enum
{
    EFctConvert_Undefined         =  0,
    EFctConvert_mMULxADDt         =  1, // y = m*x + t
    EFctConvert_mLOGxADDt         =  2, // y = m*log10(x) + t
    EFctConvert_EXP__xADDt_DIVm_  =  3, // y = 10exp((x+t)/m)
    EFctConvert_xMULr             =  4, // y = x*r
    EFctConvert_xDIVr             =  5, // y = x/r
    EFctConvert_SQRxDIVr          =  6, // y = x²/r
    EFctConvert_SQRT_xMULr_       =  7, // y = sqrt(x*r)
    EFctConvert_SQRxMULr          =  8, // y = x²*r
    EFctConvert_SQRT_xDIVr_       =  9, // y = sqrt(x/r)
    EFctConvert_mMULxADDtADDkLOGr = 10, // y = m*x + t + k*log10(r)
    EFctConvert_Count
}   EFctConvert;
```

Falls nun ein Wert zu konvertieren ist, werden die Größen „m“, „t“, und „k“ entsprechend in die jeweilige Funktion eingesetzt, die hinter dem enum-Wert als Kommentar ergänzt wurde. Die Referenzen auf die Einheiten werden benötigt, damit die Klasse *CPhysSize* sowohl innerhalb der *initialize* als auch beim Hinzufügen einer externen Umrechnungsfunktion automatisch die resultierende Umrechnungsfunktion zusammen mit den Werten für die resultierenden Parameter „m“, „t“ und „k“ ermitteln kann. Außerdem

wird zum Erzeugen des „lesbaren“ Strings sowohl das Symbol der Einheit (z.B. „V“) als auch das Formel-Symbol der Größenart (z.B. „U“) benötigt.

5.2.2. Operationen

5.2.2.1. *formatOperand(double)*

Besitzerbereich: Klasse
Sichtbarkeit: öffentlich
Beschreibung:
Wandelt eine double Wert in einen String um. Dabei wird versucht, so wenig Stellen wie möglich zu verwenden. Diese Methode wird benutzt, um Faktoren für die lineare Einheitenumrechnung innerhalb der Table-View der Beispiel-Applikation mit möglichst wenig Stellen (nur so viel, wie unbedingt notwendig) wiederzugeben, damit die Kreuztabelle nicht zu weit in die Breite gezogen wird.
Parameter:
i_fOp (IN) Datentype: double
Zu formatierender Wert.
Rückgabewert: .....Datentype: QString

5.3. Klasse *CUnit*

Basisklasse für die drei verschiedenen Einheitstypen. Da die Instanzen der Einheiten als „Blätter“ im Object Pool eingetragen werden, ist die Klasse von *QObject* abgeleitet.

5.3.1. Attribute

m_type Datentype: EUnitType
Spezifiziert den Typen der Einheit. Mögliche Werte sind „SIBase“, „Ratio“ oder „PhysUnit“.
m_pUnitGrp Datentype: CUnitGrp*
Referenz auf die Größenart, zu der die Einheit gezählt wird.
m_strName Datentype: QString
Ausgeschriebener Name der Einheit, wie z.B. „PerCent“, „Ampere“, MilliAmpere“, etc.
m_strSymbol Datentype: QString
Einheitensymbol, wie z.B. „A“, „mA“, „km“, etc.
m_iId Datentype: int
Die Id ist systemweit eindeutig und entspricht der Position der Einheit innerhalb der (linearen) Objekt-Liste des Objekt Pools.
m_bIsLogarithmic Datentype: bool
„true“ für logarithmische Einheiten.
m_fLogFactor Datentype: double
Wird nur für logarithmische Einheiten verwendet und entspricht entweder dem Wert 10.0 für Feldgrößen bzw. 20.0 für Pegelwerte. Der Faktor wird benötigt, wenn ein logarithmischer Größenwert zu einem linearen Größenwert addiert wird, was einer Multiplikation mit einem von diesem Faktor abgeleiteten Wert entspricht.
m_pNextLower und *m_pNextUpper* Datentype: CUnit*
Werden nur für lineare Einheiten verwendet, um einen Wert in seiner „bestmöglichen“ Einheit als String wiederzugeben. *m_pNextLower* verweist auf die

„nächst niedrigere“ Einheit, *m_pNextUpper* verweist auf die „nächst höhere“ Einheit derselben Größenart (z.B. ist „µm“ die nächst niedrigere Einheit von „mm“, „m“ die nächst höhere Einheit von „mm“).

5.3.2. Konstruktoren und Destruktor

5.3.2.1. *CUnit(CUnitGrp&, bool, double, const QString&, const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse. Der Konstruktor trägt die Unit-Instanz in den Unit-Objekt-Pool ein.
Parameter:
i_unitGrp (IN) Datentype: CUnitGrp
Referenz auf die Größenart, zu der die Einheit gehört.
i_bIsLogarithmic (IN) Datentype: bool
Legt fest, ob es sich bei der Einheit um eine logarithmische Einheit handelt.
i_strName (IN) Datentype: const QString&
Ausgeschriebener Name der Einheit, wie z.B. „PerCent“, „Ampere“, „Kilogramm“, „Milliwatt“, etc.
i_strSymbol (IN) Datentype: const QString&
Symbol der Einheit, wie z.B. „%“, „A“, „kg“, „mW“, etc.

5.3.2.2. *~CUnit()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Destruktor. Zerstört die Instanz und trägt die Unit-Instanz dabei wieder aus dem Objekt-Pool aus.

5.3.3. Operationen

5.3.3.1. *type()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Typen der Einheit zurück. Der Type legt fest, ob es sich bei der Instanz um eine Verhältnisgröße (Ratio), eine SI-Basiseinheit (SIBase) oder um eine „echte“, von den SI-Basiseinheiten abgeleite, physikalische Maßeinheit handelt (PhysUnit).
Rückgabewert: .....Datentype: EUnitType

5.3.3.2. *type2Str()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Typen der Einheit als String zurück.

Rückgabewert: .....Datentype: QString

5.3.3.3. *getUnitGrp()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt eine Referenz auf die Größenart zurück, zu der die Einheit gehört.
Rückgabewert: .....Datentype: CUnitGrp&

5.3.3.4. *getMainGroupName()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den übergeordneten Gruppennamen der Einheit zurück, der dem Attribut „*GroupName*“ der Klasse *CUnitGrp* entspricht. Für SI-Basiseinheiten oder Verhältnisgrößen ist der übergeordnete Gruppenname ein Leerstring. Für „echte“ physikalische Maßeinheiten der übergeordnete Gruppenname das Wissensgebiet, zu dem die Größenart gezählt wird, wie z.B. „Geometry“, „Electricity“ oder „Kinematics“.
Rückgabewert: .....Datentype: QString

5.3.3.5. *getSubGroupName()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Gruppennamen der Einheit zurück, der dem Attribut „Name“ der Klasse *CUnitGrp* entspricht. Für SI-Basiseinheiten ist der Gruppenname gleich „SIBase“, für Verhältnisgrößen ist der Gruppenname „Ratio“. Für „echte“ physikalische Maßeinheiten ist der Gruppenname die Größenart, zu der die Einheit gehört, wie z.B. „Current“, „Voltage“, „Length“, etc.
Rückgabewert: .....Datentype: QString

5.3.3.6. *getName()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den ausgeschriebenen Namen der Einheit zurück, wie z.B. „PerCent“, „Ampere“, „Kilogramm“, „Milliwatt“, etc.
Rückgabewert: .....Datentype: QString

5.3.3.7. *getSymbol()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt das Symbol der Einheit zurück, wie z.B. „%“, „A“, „kg“, „mW“, etc.
Rückgabewert: .....Datentype: QString

5.3.3.8. getId()

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Instanzen der Klasse *CUnit* werden beim Konstruieren in einen Objektpool eingetragen und sind so sowohl über ihre Namen als auch über ihre systemweit eindeutige ID erreichbar. Diese Methode gibt die eindeutige ID der Einheit zurück.
Rückgabewert: .....Datentype: int

5.3.3.9. *operator ==*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse *CUnit*. Eine Gleichheit der Instanzen besteht dann, wenn ihre Speicheradresse identisch ist.
Rückgabewert: .....Datentype: bool

5.3.3.10. *operator !=*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse *CUnit*. Eine Gleichheit der Instanzen besteht dann, wenn ihre Speicheradresse identisch ist.
Rückgabewert: .....Datentype: bool

5.3.3.11. *isLogarithmic()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt an, ob es sich um eine logarithmische Einheit handelt.
Rückgabewert: .....Datentype: bool

5.3.3.12. *getLogaritmicFactor()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Faktor zurück, der benötigt wird, wenn ein logarithmischer Größenwert zu einem linearen Größenwert addiert wird, was einer Multiplikation mit einem von diesem Faktor abgeleiteten Wert entspricht.
Rückgabewert: .....Datentype: double

5.3.3.13. *isConvertible(const CUnit&, double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Testet, ob die Einheit in die definierte Zieleinheit konvertiert werden kann.
Parameter:
i_unitDst (IN) Datentype: CUnit&
Zieleinheit, in die der Wert zu konvertieren wäre.
i_fVal (IN) Datentype: double
Wert, der zu konvertieren wäre (Default = 1.0)
Rückgabewert: .....Datentype: bool

5.3.3.14. *convertValue(double, const CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Konvertiert den übergebenen Wert in die definierte Zieleinheit.
Parameter:
i_fVal (IN) Datentype: double
Zu konvertierender Wert.
i_unitDst (IN) Datentype: CUnit&
Zieleinheit, in die der Wert zu konvertieren ist.
Rückgabewert: .....Datentype: double

5.3.3.15. *setNextLowerHigherUnits(CUnit*, CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt die nächst höhere und die nächst niedere Einheit fest. Nur für lineare Einheiten sinnvoll. Die Verkettung der Einheit über *NextLower* und *NextHigher* wird benötigt, um einen Wert in seiner „bestmöglichen“ Einheit als String wiederzugeben. Die „*initialize*“ Methode der Klasse *CPhysSize* kann die Verkettung der Einheiten automatisch vornehmen, so fern sich die Einheiten immer um den Faktor 10^3 unterscheiden. Ist diese automatische Verkettung einmal nicht anwendbar, kann die Verkettung explizit definiert werden. Dies ist z.B. bei Längen-Einheiten sinnvoll, wenn zusätzlich zu den metrischen auch die britischen Maßangaben wie Inch und Feet benutzt werden oder um „cm“ und „dm“ aus der „*findBestUnit*“ Verkettung auszuschließen.

m_pNextLower verweist auf die „nächst niedrigere“ Einheit, *m_pNextUpper* verweist auf die „nächst höhere“ Einheit derselben Größenart (z.B. ist „µm“ die nächst niedrigere Einheit von „mm“, „m“ die nächst höhere Einheit von „mm“).

Parameter:
i_pNextLower (IN) Datentype: CUnit*
Verweis auf die nächst niedere Einheit (z.B. ist „µm“ die nächst niedere Einheit von „mm“).
i_pNextHigher (IN) Datentype: CUnit*
Verweis auf die nächst höhere Einheit (z.B. ist „m“ die nächst höhere Einheit von „mm“).

5.3.3.16. *getNextLowerUnit()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die nächst niedere Einheit zurück.
Rückgabewert: .....Datentype: CUnit*

5.3.3.17. *getNextHigherUnit()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die nächst höhere Einheit zurück.
Rückgabewert: .....Datentype: CUnit*

5.3.3.18. *findBestUnit(double, double*, int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Sucht die bestmögliche Einheit für den übergebenen Wert und gibt (optional) den Wert in der bestmöglichen Einheit zurück.
Beispiele (Größenart Länge mit „m“, „mm“, µm“):

Ausgangswert	Wert in bestmöglicher Einheit
123456789 µm	123,456789 m
0,000000012345 m	0,012345 µm

Parameter:
i_fVal (IN) Datentype: double
Zu konvertierender Wert.
o_pfVal (IN) Datentype: double*
Falls ungleich NULL, wird hier der in die bestmögliche Einheit konvertierte Wert zurückgegeben.
iDigitsLeadingMax (IN) Datentype: int
Über diesen Wert kann die Art und Weise geändert werden, wie die Methode entscheidet, was die bestmögliche Einheit ist. Per Default ist für die Methode die bestmögliche Einheit, wenn der übergebene Wert mit mindestens einer, maximal mit drei Vorkommastellen wiedergegeben werden kann oder mit einer Anzahl von Vorkomma- und Nachkommastellen, die dieser Vorgabe am nächsten kommt.
Rückgabewert: .....Datentype: CUnit*

5.4. Klasse *CUnitGrp*

Basisklasse für die Klassen zur Gruppierung der Einheiten. Da die Instanzen der Einheiten als „Äste“ im Objekt Pool eingetragen werden, ist die Klasse von *QObject* abgeleitet.

5.4.1. Attribute

m_type Datentype: EUnitType
Spezifiziert den Typen der Einheit. Mögliche Werte sind „SIBase“, „Ratio“ oder „PhysUnit“.

m_strGroupName Datentype: QString
Falls die Klasse SI-Basiseinheiten oder Verhältnisgrößen verwaltet, ist hier ein Leerstring eingetragen. Falls die Klasse „echte“ Maßeinheiten verwaltet, ist hier der Name des Wissensgebiets eingetragen, zu dem die Größenart gezählt wird, wie z.B. „Geometry“, „Electricity“ oder „Kinematics“.

m_strName Datentype: QString
Falls die Klasse SI-Basiseinheiten verwaltet, steht hier „SI-Base“, im Falle der Verhältnisgrößen „Ratio“. Falls die Klasse „echte“ Maßeinheiten verwaltet, steht hier der Name der Größenart, wie z.B. „Current“, „Voltage“, „Length“, etc.

m_pObjPoolTreeEntry Datentype: CObjPoolTreeEntry
Referenz auf den zugehörigen Eintrag im Objekt Pool. Das „TreeEntry“-Attribut verweist auf den zugehörigen Ast innerhalb der Baumstruktur, der dem Gruppeneintrag entspricht. Wird benötigt, um durch den Objekt Pool zu „wandern“ und so die „Childs“ – also die Einheiten – der Größenart zu finden.

5.4.2. Konstruktoren und Destruktor

5.4.2.1. *CUnitGrp(EUnitType, const QString&, const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt eine Instanz der Klasse. Der Konstruktor trägt die Gruppen-Instanz in den Unit-Objekt-Pool ein.

Parameter:

i_type (IN) Datentype: EUnitType
Legt den Typen der Größenart und damit auch den Typen für die Einheiten der Größenart fest. Mögliche Werte sind „SIBase“, „Ratio“ oder „PhysUnit“.

i_strGroupName (IN) Datentype: const QString&
Falls die Klasse SI-Basiseinheiten oder Verhältnisgrößen verwaltet, ist hier ein Leerstring eingetragen. Falls die Klasse „echte“ Maßeinheiten verwaltet, ist hier der Name des Wissensgebiets eingetragen, zu dem die Größenart gezählt wird, wie z.B. „Geometry“, „Electricity“ oder „Kinematics“.

i_strName (IN) Datentype: const QString&
Falls die Klasse SI-Basiseinheiten verwaltet, steht hier „SI-Base“, im Falle der Verhältnisgrößen „Ratio“. Falls die Klasse „echte“ Maßeinheiten verwaltet, steht hier der Name der Größenart, wie z.B. „Current“, „Voltage“, „Length“, etc.

5.4.2.2. *~CUnit()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Destruktor. Zerstört die Instanz und trägt die Gruppen-Instanz dabei wieder aus dem Objekt-Pool aus.

5.4.3. Operationen

5.4.3.1. *type()*

Besitzerbereich: Instanz

Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Typen der Größenart zurück. Der Type legt fest, ob es sich bei der Instanz um eine Verhältnisgröße (Ratio), eine SI-Basiseinheit (SIBase) oder um eine „echte“, von den SI-Basiseinheiten abgeleite, physikalische Maßeinheit handelt (PhysSize).
Rückgabewert: .....Datentype: EUnitType

5.4.3.2. *type2Str()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Typen der Einheit als String zurück.
Rückgabewert: .....Datentype: QString

5.4.3.3. *getGroupName()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Gruppennamen der Einheit zurück. Für SI-Basiseinheiten oder Verhältnisgrößen ist der übergeordnete Gruppenname ein Leerstring. Für „echte“ physikalische Größenarten ist der übergeordnete Gruppenname das Wissensgebiet, zu dem die Größenart gezählt wird, wie z.B. „Geometry“, „Electricity“ oder „Kinematics“.
Rückgabewert: .....Datentype: QString

5.4.3.4. *getName()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Namen der Größenart zurück. Für SI-Basiseinheiten ist der Gruppenname gleich „SIBase“, für Verhältnisgrößen ist der Gruppenname „Ratio“. Für „echte“ physikalische Größenarten ist der Gruppenname die Größenart, zu der die Einheit gehört, wie z.B. „Current“, „Voltage“, „Length“, etc.
Rückgabewert: .....Datentype: QString

5.4.3.5. *getUnitCount()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Liefert die Anzahl der Einheiten zurück, die der Größenart zugewiesen wurden.
Rückgabewert: Datentype: unsigned int

5.4.3.6. *getUnit(unsigned int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:

Liefert die Adresse auf die Einheiten-Instanz zurück. Wird in Verbindung mit *getUnitCount* verwendet, um über einen Index Zugriff auf die Einheiten der Größenart zuzugreifen.

Parameter:

i_udx (IN) Datentype: unsigned int

Index der Einheit, die zurückgegeben werden soll. Entspricht dem „RowIdx“ des „Objekt-Pool-Tree“ Eintrags der Größenart.

Rückgabewert: Datentype: CUnit*

5.4.3.7. *findUnit(const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Sucht die Einheit anhand des übergebenen Strings. Dabei wird sowohl auf Name als auch auf Symbol geprüft. Die Methode liefert also z.B. für „Meter“ und „m“ dieselbe Einheiten-Instanz zurück.

Parameter:

i_strSymbolOrName (IN) Datentype: const QString&

Name oder Symbol der Einheit, nach der gesucht werden soll.

Rückgabewert: Datentype: CUnit*

5.4.3.8. *findUnitBySymbol(const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Sucht die Einheit anhand des übergebenen Symbols für die Einheit.

Parameter:

i_strSymbol (IN) Datentype: const QString&

Symbol der Einheit, nach der gesucht werden soll.

Rückgabewert: Datentype: CUnit*

5.4.3.9. *findUnitByName(const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Sucht die Einheit anhand des übergebenen Namens für die Einheit.

Parameter:

i_strName (IN) Datentype: const QString&

Ausgeschriebener Name der Einheit (z.B. „Meter“), nach der gesucht werden soll.

Rückgabewert: Datentype: CUnit*

5.4.3.10. *getObjPoolTreeEntry()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Gibt den Tree-Entry Eintrag zurück, in dem die Größenart innerhalb des Unit-Objekt-Pools eingetragen wurde.
Rückgabewert: Datentype: CObjPoolTreeEntry*

5.4.3.11. *operator ==*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Speicheradresse identisch ist.
Rückgabewert: ..... Datentype: bool

5.4.3.12. *operator !=*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Speicheradresse identisch ist.
Rückgabewert: ..... Datentype: bool

5.5. Klasse *CPhysUnit*

Instanzen dieser Klasse entsprechen den „echten“ Maßeinheiten physikalischer Größen wie z.B. „MilliAmpere“, „Kilometer“, „Meter“, „Kilogramm“, etc. Welche Einheiten verwendet werden, ist spezifisch für jede Anwendung. Deshalb werden diese Einheiten erst zusammen mit der jeweiligen Applikation erzeugt.

5.5.1. Attribute

m_pPhysSize Datentype: CPhysSize*
Einheiten werden innerhalb einer Instanz der Klasse *CPhysSize* verwaltet. Zu welcher physikalischen Größenart die Einheit gehört, wird über dieses Attribut festgelegt.

m_pPhysUnitSI Datentype: CPhysUnit*
Innerhalb jeder Größenart gibt es eine SI-Einheit. Für die elektrische Stromstärke wäre dies z.B. „Ampere“, für die Länge „Meter“ usw. Die SI-Einheit muss für jede Größenart angelegt werden. Eine Referenz auf die SI-Einheit wird in diesem Attribut gespeichert.

m_strPrefix Datentype: QString
Lineare Einheiten, die sich von der SI-Einheit um einen dekadischen Faktor unterscheiden, werden eindeutig durch den Prefix bestimmt (z.B. „m“, „k“, „µ“, „G“). Aus dem Prefix lässt sich automatisch der dekadische Faktor ableiten (z.B. 10^{-3} für „m“, 10^{3} für „k“, etc.).

m_bInitialized Datentype: bool
Die Größenarten und ihre Einheiten müssen durch den „*initialize*“ Aufruf der Klasse *CPhysSize* initialisiert werden. Durch den „*initialize*“ Aufruf wird das Flag „*Initialized*“ auf *true* gesetzt. Erst eine vollständig „initialisierte“ Größenart und Einheit ist in der Lage, Einheiten umzurechnen. Für weitere Informationen siehe *initialize* Methode der Klasse *CPhysSize*.

m_iPhysSizeTreeEntryRowIdx Datentype: int
Instanzen der Klasse *CPhysUnit* werden in den Objekt-Pool als „Kinder“ der Klasse *CPhysSize* eingetragen. Damit sind sie über einen Row-Index (beginnend mit 0) als Child-Objekte von *CPhysSize* adressierbar. Dabei entspricht die Anzahl der Child-Objekte (*RowCount*) genau der Anzahl der Maßeinheiten der physikalischen Größenart. Die interne Umrechnungungstabelle der Maßeinheit wird mit *RowCount* Elementen allokiert und der „*TreeEntryRowIdx*“ der Instanz entspricht dem Index-Wert der Instanz innerhalb dieser internen Umrechnungstabelle. Oder mit anderen Worten: an diesem Index befindet sich die Anweisung, wie die Maßeinheit in sich selbst umzurechnen wäre (also eine „neutrale“ Umrechnungsfunktion wie z.B. eine lineare Geradengleichung mit Faktor m = 1 und Summand t = 0).

m_fctConvertFromSIUnit Datentype: CFctConvert
Definiert die Umrechnungsfunktion die angewendet werden muss, um einen Größenwert, der in der SI-Einheit der Größenart vorliegt, in diese Maßeinheit umzurechnen.

m_fctConvertIntoSIUnit Datentype: CFctConvert
Definiert die Umrechnungsfunktion die angewendet werden muss, um diese Maßeinheit in die SI-Einheit der Größenart umzurechnen.

m_arFctConvertsInternal Datentype: CFctConvert*
Liste mit Funktionen, um diese Maßeinheit in andere Einheiten derselben Größenart umzurechnen (interne Umrechnungstabelle). Die Liste enthält genau ein Element für jede Maßeinheit der Größenart, wobei das Attribut *TreeEntryRowIdx* den Index in das Feld für die Maßeinheit festlegt. Besitzt z.B. die Einheit „kV“ den *RowIdx* = 5 und muss ein Größenwert von „µV“ nach „kV“ konvertiert werden, steht die anzuwendende Umrechnungsfunktion am Element mit dem Index = 5 innerhalb der Umrechnungstabelle der Maßeinheit „µV“. Um das Debuggen zu erleichtern wurde auf die Verwendung eines Template Datencontainers verzichtet.

m_uFctConvertsInternalCount Datentype: unsigned int
Länge der internen Umrechnungstabelle. Da die Länge der Umrechnungstabelle nicht dynamisch verändert werden kann, entspricht die Länge der Umrechnungstabelle immer gleich der Anzahl der belegten Elemente.

m_arFctConvertsExternal Datentype: CFctConvert*
Liste mit Funktionen, um diese Maßeinheit in eine (beliebig) andere Maßeinheit einer anderen Größenart umzurechnen (externe Umrechnungstabelle). Externen Umrechnungsfunktionen werden dynamisch zur Laufzeit hinzugefügt. Um die anzuwendende Umrechnungsfunktion zu finden, muss die Liste anhand der gewünschten Ziel-Einheit durchsucht werden. Um das Debuggen zu erleichtern wurde auf die Verwendung eines Template Datencontainers verzichtet.

m_uFctConvertsExternalCount Datentype: unsigned int
Anzahl der belegten Elemente innerhalb der externen Umrechnungstabelle. Diese ist immer kleiner oder gleich der aktuellen Länge der Umrechnungstabelle.

m_uFctConvertsExternalArrLen Datentype: unsigned int
Länge der externen Umrechnungstabelle. Diese ist immer größer oder gleich der Anzahl der belegten Elemente innerhalb der Umrechnungstabelle. Soll der Tabelle ein neues Element hinzugefügt werden, sind aber bereits alle Elemente belegt, wird die Tabelle vergrößert. Dabei wird die Tabelle nicht nur um ein Element vergrößert, sondern es wird versucht, doppelt so viele Elemente zusätzlich zu reservieren, wie die Tabelle aktuell besaß. Dabei ist die Anzahl der zusätzlich zu reservierenden Elemente allerdings auf eine sinnvolle, maximale Anzahl begrenzt (nicht jedoch die Gesamtlänge der Tabelle).

5.5.2. Konstruktoren und Destruktor

5.5.2.1. *CPhysUnit(CPhysSize&, const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt eine Instanz der Klasse für lineare Einheiten, deren Umrechnungsfaktor in die SI-Unit eine Zehnerpotenz ist (z.B. „km“, „mV“, etc.). Der Konstruktor der Basisklasse CUnit wird implizit so aufgerufen, dass eine Instanz vom Einheiten-Type „PhysUnit“ erzeugt wird.

Parameter:

i_physSize (IN) Datentype: CPhysSize&
Referenz auf die Größenart, zu der die Einheit gehört.

i_strPrefix (IN) Datentype: const QString&
Präfix-String des Umrechnungsfaktor der Einheit (z.B. „m“ für Milli, „k“ für Kilo, „µ“ für Micro, „G“ für Giga, etc.).

5.5.2.2. *CPhysUnit(CPhysSize&, bool, const QString&, const QString&, double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt eine Instanz der Klasse für lineare Einheiten, deren Umrechnungsfaktor nicht dekadisch ist (z.B. „Inch“, „Zoll“, „Minuten“, „Tage“, etc.) oder eine Instanz der Klasse für logarithmische Einheiten. Der Konstruktor der Basisklasse *CUnit* wird implizit so aufgerufen, dass eine Instanz vom Einheiten-Type „PhysUnit“ erzeugt wird.

Parameter:

i_physSize (IN) Datentype: CPhysSize&
Referenz auf die Größenart, zu der die Einheit gehört.

i_bIsLogarithmic (IN) Datentype: bool
Legt fest, ob es sich bei der Einheit um eine logarithmische Einheit handelt.

i_strName (IN) Datentype: const QString&
Ausgeschriebener Name der Einheit, wie z.B. „PerCent“, „Ampere“, „Kilogramm“, „Milliwatt“, etc.

i_strSymbol (IN) Datentype: const QString&
Symbol der Einheit, wie z.B. „%“, „A“, „kg“, „mW“, etc.

i_fMFromBaseOrRefVal (IN) Datentype: double
Im Falle einer nicht logarithmischen Einheit ist hier der Faktor zu übergeben, mit dem Werte aus der SI Basis-Einheit in diese Einheit umzurechnen sind (z.B. 10^{-3} für „kilo“, 10^{6} für Micro).
Im Falle einer logarithmischen Einheit ist hier der Referenzwert zu übergeben (also z.B. 10^{-3} für Milli-Watt).

5.5.2.3. *~CPhysUnit()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Destruktor. Zerstört die Instanz und trägt die Unit-Instanz dabei wieder aus dem Objekt-Pool aus.

5.5.3. Operationen

5.5.3.1. *operator ==*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Speicheradresse identisch ist.
Rückgabewert: .....Datentype: bool

5.5.3.2. *operator !=*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Speicheradresse identisch ist.
Rückgabewert: .....Datentype: bool

5.5.3.3. *getPhysSize()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt eine Referenz auf die Größenart zurück, zu der die Einheit gehört.
Rückgabewert: .....Datentype: CPhysSize&

5.5.3.4. *getSIUnit()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die Adresse der SI Einheit der Größenart zurück.
Rückgabewert: .....Datentype: CPhysUnit*

5.5.3.5. *getPrefixStr()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Präfix-String des Umrechnungsfaktor der Einheit zurück (z.B. „m“ für Milli, „k“ für Kilo, „µ“ für Micro, „G“ für Giga, etc.).
Rückgabewert: .....Datentype: QString

5.5.3.6. *isConvertible(const CUnit&, double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:

Überschreibt die entsprechende virtuelle Methode der Basisklasse. Testet, ob die Einheit in die definierte Zieleinheit konvertiert werden kann.

Parameter:

i_unitDst (IN) Datentype: CUnit&

Zieleinheit, in die der Wert zu konvertieren wäre.

i_fVal (IN) Datentype: double

Wert, der zu konvertieren wäre (Default = 1.0)

Rückgabewert: .....Datentype: bool

5.5.3.7. *convertValue(double, const CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Überschreibt die entsprechende virtuelle Methode der Basisklasse. Konvertiert den übergebenen Wert aus der Einheit, für die die Methode aufgerufen wurde, in die definierte Zieleinheit. Beispiel:

```
double     fVal_V = 0.1;
CPhysUnit& physUnitV = Electricity::Voltage().MilliVolt();
CPhysUnit& physUnitmV = Electricity::Voltage().MilliVolt();
double     fVal_mV = physUnitV.convertValue(fVal_V,physUnitmV);
Ergebnis:  fVal_mV = 100.0;
```

Parameter:

i_fVal (IN) Datentype: double

Zu konvertierender Wert in der Einheit, für die die Methode aufgerufen wird.

i_unitDst (IN) Datentype: CUnit&

Zieleinheit, in die der Wert zu konvertieren ist.

Rückgabewert: Datentype: double

Wert in der Zieleinheit.

5.5.3.8. *convertFromSIUnit(double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Konvertiert den übergebenen Wert aus der SI Einheit der Größenart in die Einheit, für die die Methode aufgerufen wird. Beispiel:

```
double     fVal_V = 0.1;
CPhysUnit& physUnitmV = Electricity::Voltage().MilliVolt();
double     fVal_mV = physUnitmV.convertFromSIUnit(fVal_V);
Ergebnis:  fVal_mV = 100.0;
```

Parameter:

i_fVal (IN) Datentype: double

Zu konvertierender Wert in der SI Einheit der Größenart.

Rückgabewert: Datentype: double

Wert in der Einheit, für die die Methode aufgerufen wurde.

5.5.3.9. *convertIntoSIUnit(double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Konvertiert den übergebenen Wert aus der Einheit, für die die Methode aufgerufen wird, in die SI Einheit der Größenart. Beispiel:

```
double     fVal_mV = 100.0;
CPhysUnit& physUnitmV = Electricity::Voltage().MilliVolt();
double     fVal_V = physUnitmV.convertIntoSIUnit(fVal_mV);
Ergebnis:  fVal_V = 0.1;
```

Parameter:

i_fVal (IN) Datentype: double
Zu konvertierender Wert in der Einheit, für die die Methode aufgerufen wird.

Rückgabewert: Datentype: double
Wert in der SI Einheit der Größenart.

5.5.3.10. *convertValue(double, const CPhysUnit&, double, const CPhysUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Konvertiert den Wert aus der Einheit, für den die Methode aufgerufen wird, in die gewünschte Zieleinheit unter Verwendung des definierten Referenzwertes und seiner Einheit. Beispiel:

```
double     fVal_mV     = 100.0;
double     fRef_Ohm    = 1.0e3;
CPhysUnit& physUnitmV  = Electricity::Voltage().MilliVolt();
CPhysUnit& physUnitmA  = Electricity::Current().MilliAmpere();
CPhysUnit& physUnitOhm = Electricity::Resistance().Ohm();
double     fVal_mA     = physUnitmV.convertValue(
                             /* fVal    */ fVal_mV,
                             /* unitDst */ physUnitmA,
                             /* fRef    */ fRef_Ohm,
                             /* unitRef */ physUnitOhm );
Ergebnis:  fVal_mA = 0.1;
```

Parameter:

I_fVal (IN) Datentype: double
Zu konvertierender Wert in der Einheit, für die die Methode aufgerufen wird.

i_unitDst (IN) Datentype: const CPhysUnit&
Gewünschte Zieleinheit.

I_fRef (IN) Datentype: double
Referenzwert, der bei der Umrechnung zu berücksichtigen ist.

i_unitRef (IN) Datentype: const CPhysUnit&
Einheit des Referenzwertes.

Rückgabewert: Datentype: double
Wert in der gewünschten Zieleinheit.

5.5.3.11. *convertValue(double, const CPhysUnit&, double, const CPhysUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: geschützt

Beschreibung:

Konvertiert den Wert aus der Einheit, für den die Methode aufgerufen wird, in die gewünschte Zieleinheit unter Verwendung des definierten Referenzwertes und seiner Einheit. Im Unterschied zur öffentlichen *convertValue* Methode wird die Referenzeinheit als Pointer übergeben und ist ein optionaler Parameter. Diese Methode wird von allen anderen *convertValue* Methode aufgerufen, um beinhaltet die eigentliche Konvertierungsroutine.

Parameter:
I_fVal (IN) Datentype: double
Zu konvertierender Wert in der Einheit, für die die Methode aufgerufen wird.
i_unitDst (IN) Datentype: const CPhysUnit&
Gewünschte Zieleinheit.
I_fRef (IN) Datentype: double
Referenzwert, der bei der Umrechnung zu berücksichtigen ist.
i_unitRef (IN) Datentype: const CPhysUnit*
Einheit des Referenzwertes.
Rückgabewert: Datentype: double
Wert in der gewünschten Zieleinheit.

5.5.3.12. *getFctConvertFromSIUnit*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die Beschreibung der Funktion zurück, mit der die SI-Einheit der Größenart in diese Einheit umzurechnen ist.
Rückgabewert: ..... Datentype: CFctConvert&

5.5.3.13. *getFctConvertFromSIUnitName*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die Umrechnungsfunktion, mit der die SI-Einheit der Größenart in diese Einheit umzurechnen ist, in Form eines Strings zurück (z.B. zur Darstellung der Umrechnungsfunktionen innerhalb der Beispiel- und Testapplikation des PhysVal-Subsystems)
Rückgabewert: ..... Datentype: QString

5.5.3.14. *getFctConvertIntoSIUnit*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die Beschreibung der Funktion zurück, mit der die Einheit in die SI-Einheit der Größenart umzurechnen ist.
Rückgabewert: ..... Datentype: CFctConvert&

5.5.3.15. *getFctConvertIntoSIUnitName*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die Umrechnungsfunktion, mit der die Einheit in die SI-Einheit der Größenart umzurechnen ist, in Form eines Strings zurück (z.B. zur Darstellung der Umrechnungsfunktionen innerhalb der Beispiel- und Testapplikation des PhysVal-Subsystems)

Rückgabewert: .....Datentype: QString

5.5.3.16. *getFctConvertsInternalCount*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die Anzahl der internen Umrechnungsfunktionen zurück. Wurden die internen Umrechnungsfunktionen über die *initialize* Methode der Klasse *CPhysSize* automatisch ermittelt, entspricht diese der Anzahl der Einheiten der Größenart, da die *initialize* Methode für jede Einheit eine Umrechnungsfunktion in jede andere Einheit generiert.
Rückgabewert: .....Datentype: unsigned int

5.5.3.17. *findFctConvertInternalIdx(CPhysUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Umrechnungsfunktionen werden in einer Liste gehalten. Mit dieser Methode erhalten Sie den Index innerhalb dieser Liste, unter dem die Umrechnungsfunktion in die übergebene Zieleinheit abgelegt ist.
Parameter:
i_physUnitDst (IN) Datentype: CPhysUnit&
Einheit, für die der Index der Umrechnungsfunktion ermittelt werden soll.
Rückgabewert: .....Datentype: int

5.5.3.18. *getFctConvertInternal(unsigned int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Umrechnungsfunktionen werden in einer Liste gehalten. Die Methode liefert die Umrechnungsfunktion am entsprechenden Index in der Liste.
Parameter:
i_udx (IN) Datentype: unsigned int
Index in die Liste der Umrechnungsfunktionen.
Rückgabewert: .....Datentype: CFctConvert*

5.5.3.19. *findFctConvertInternal(const CPhysUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Umrechnungsfunktionen werden in einer Liste gehalten. Mit dieser Methode erhalten Sie die Beschreibung der Umrechnungsfunktion, mit der die Einheit in die übergebene Zieleinheit umgerechnet wird.
Parameter:
i_physUnitDst (IN) Datentype: CPhysUnit*
Einheit, für die die Umrechnungsfunktion ermittelt werden soll.
Rückgabewert: .....Datentype: CFctConvert*

5.5.3.20. *findFctConvertInternalName(const CPhysUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Umrechnungsfunktionen werden in einer Liste gehalten. Zudem wird für jede Umrechnungsfunktion ein String abgelegt, der die mathematische Funktion in „lesbarer“ Form beschreibt. Mit dieser Methode erhalten Sie diese lesbare Form der Umrechnungsfunktion als String, mit der die Einheit in die übergebene Zieleinheit umgerechnet wird.
Parameter:
i_physUnitDst (IN) Datentype: CPhysUnit*
Einheit, für die die Umrechnungsfunktion ermittelt werden soll.
Rückgabewert: .....Datentype: QString

5.5.3.21. *getFctConvertInternalName(unsigned int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Umrechnungsfunktionen werden in einer Liste gehalten. Zudem wird für jede Umrechnungsfunktion ein String abgelegt, der die mathematische Funktion in „lesbarer“ Form beschreibt. Mit dieser Methode erhalten Sie diese lesbare Form der Umrechnungsfunktion als String, die sich am entsprechenden Index innerhalb der Liste der Umrechnungsfunktionen befindet.
Parameter:
i_udx (IN) Datentype: unsigned int
Index in die Liste der Umrechnungsfunktionen.
Rückgabewert: .....Datentype: QString

5.5.3.22. *addFctConvertExternal(CPhysUnit&, CPhysUnit&, EFctConvert)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Um eine Einheit in eine Einheit einer anderen Größenart umzurechnen, werden „externe“ Umrechnungsfunktionen verwendet, die zur Umrechnung eine Referenzeinheit benötigen. Der „normale“ Weg, eine Einheit in eine Einheit einer anderen Größenart umzurechnen ist, die Quell-Einheit zunächst in deren SI Einheit umzurechnen, anschließend den Wert in die SI Einheit der Zieleinheit umzurechnen und im dritten Schritt den Wert in die Zieleinheit zu konvertieren. Bei der Umrechnung einer logarithmischen Quell-Einheit in eine logarithmische Zieleinheit sind diese drei Rechenschritte unnötig, da zur Umrechnung lediglich konstante Summanden zu addieren sind.
Über die „*addFctConvertExternal*“ kann eine direkte Umrechnungsfunktion in die gewünschte Zieleinheit unter Verwendung der definierten Referenzeinheit festgelegt werden. Allerdings gelten folgende Einschränkungen:
1. Sie kann für logarithmische Quell-Einheiten aufgerufen werden, wenn die Zieleinheit ebenfalls logarithmisch ist.

2. Ansonsten kann sie nur für die SI Einheit einer Größenart aufgerufen werden und auch nur dann, wenn sich die SI-Einheit mit einer mathematischen Funktion in die Zieleinheit konvertieren lässt, die nicht die die Parameter „m“ und „t“ benötigt. Dies sind die Funktionen „x*r“, „x²/r“, „x/r“, „√(x/r)“, „x²*r“, „√(x/r)“. Diese mathematischen Funktionen reichten bislang aus, um alle wirklich benötigten Umrechnungen zwischen unterschiedlichen Größenarten zu realisieren.

Die Einschränkungen gelten wegen der implementierten Automatismen, die bei der Konvertierung der Einheiten zum Tragen kommen, um die resultierende Umrechnungsfunktion mit den anzuwendenden Konstanten „m“, „t“, und „k“ aus den Umrechnungsfunktionen der Quell- und Ziel-Einheit dynamisch zu ermitteln.

Anmerkung:
In der Regel werden Sie nicht in die Verlegenheit kommen, selbst einer Einheit eine externe Umrechnungsfunktion hinzufügen zu müssen. Denn diese Aufgabe erledigt die Klasse *CPhysSize* automatisch, wenn dort über „*addFctConvert*“ eine Umrechnungsfunktion in eine andere Größenart festgelegt wird. Wie ferner bereits erwähnt, war es bislang nicht notwendig, „komplexere“ Umrechnungsfunktionen festlegen zu können. Sollte dies einmal notwendig sein, würde es sich anbieten, die Möglichkeit zu implementieren, benutzerdefinierte Umrechnungsfunktionen anlegen zu können.

Parameter:

i_physUnitDst (IN) Datentype: CPhysUnit&
Einheit, in die die Quelleinheit umzurechnen ist.

i_physUnitRef (IN) Datentype: CPhysUnit&
Referenzeinheit, die bei der Umrechnung zu verwenden ist. Der Referenzwert wird von der Größenart der Referenzeinheit ausgelesen.

i_fctConvert (IN) Datentype: EFctConvert
Mathematische Umrechnungsfunktion, um die Quelleinheit in die Zieleinheit umzurechnen. Dabei gelten die oben beschriebenen Einschränkungen.

5.5.3.23. *getFctConvertsExternalCount*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Gibt die Anzahl der externen Umrechnungsfunktionen zurück.

Rückgabewert: .....Datentype: unsigned int

5.5.3.24. *findFctConvertExternalIdx(CPhysUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Die Umrechnungsfunktionen werden in einer Liste gehalten. Mit dieser Methode erhalten Sie den Index innerhalb dieser Liste, unter dem die Umrechnungsfunktion in die übergebene Zieleinheit abgelegt ist.

Parameter:

i_physUnitDst (IN) Datentype: CPhysUnit&
Einheit, für die der Index der Umrechnungsfunktion ermittelt werden soll.

Rückgabewert: .....Datentype: int

5.5.3.25. *getFctConvertExternal(unsigned int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Umrechnungsfunktionen werden in einer Liste gehalten. Die Methoden liefert die Umrechnungsfunktion am entsprechenden Index in der Liste.
Parameter:
i_udx (IN) Datentype: unsigned int
Index in die Liste der Umrechnungsfunktionen.
Rückgabewert: ..... Datentype: CFctConvert*

5.5.3.26. *findFctConvertExternal(const CPhysUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Umrechnungsfunktionen werden in einer Liste gehalten. Mit dieser Methode erhalten Sie die Beschreibung der Umrechnungsfunktion, mit der die Einheit in die übergebene Zieleinheit umgerechnet wird.
Parameter:
i_physUnitDst (IN) Datentype: CPhysUnit*
Einheit, für die die Umrechnungsfunktion ermittelt werden soll.
Rückgabewert: ..... Datentype: CFctConvert*

5.5.3.27. *findFctConvertExternalName(const CPhysUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Umrechnungsfunktionen werden in einer Liste gehalten. Zudem wird für jede Umrechnungsfunktion ein String abgelegt, der die mathematische Funktion in „lesbarer“ Form beschreibt. Mit dieser Methode erhalten Sie diese lesbare Form der Umrechnungsfunktion als String, mit der die Einheit in die übergebene Zieleinheit umgerechnet wird.
Parameter:
i_physUnitDst (IN) Datentype: CPhysUnit*
Einheit, für die die Umrechnungsfunktion ermittelt werden soll.
Rückgabewert: ..... Datentype: QString

5.5.3.28. *getFctConvertExternalName(unsigned int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Umrechnungsfunktionen werden in einer Liste gehalten. Zudem wird für jede Umrechnungsfunktion ein String abgelegt, der die mathematische Funktion in „lesbarer“ Form beschreibt. Mit dieser Methode erhalten Sie diese lesbare Form der Umrechnungsfunktion als String, die sich am entsprechenden Index innerhalb der Liste der Umrechnungsfunktionen befindet.
Parameter:

i_udx (IN) Datentype: unsigned int
Index in die Liste der Umrechnungsfunktionen.
Rückgabewert: .....Datentype: QString

5.6. Klasse *CPhysSize*

Instanzen dieser Klasse entsprechen einer Größenart, die die zugehörigen, „echten“ Maßeinheiten zu einer Gruppe zusammenfasst. Einheiten innerhalb ein und derselben Größenart unterscheiden sich meist nur durch den Faktor, dem „Prefix“ vor dem SI-Einheitensymbol.

5.6.1. Attribute

m_strSIUnitName Datentype: QString
Innerhalb jede Größenart gibt es eine SI-Einheit. Für die elektrische Stromstärke wäre dies z.B. „Ampere“, für die Länge „Meter“ usw. Die SI-Einheit muss für jede Größenart definiert werden. Der (ausgeschriebene) Name (z.B. „MilliAmpere“) wird in diesem Attribut gespeichert.

m_strSIUnitSymbol Datentype: QString
Innerhalb jede Größenart gibt es eine SI-Einheit. Für die elektrische Stromstärke wäre dies z.B. „Ampere“, für die Länge „Meter“ usw. Die SI-Einheit muss für jede Größenart definiert werden. Das Symbol (z.B. „mA“) wird in diesem Attribut gespeichert.

m_pPhysUnitSI Datentype: CPhysUnit*
Innerhalb jede Größenart gibt es eine SI-Einheit. Für die elektrische Stromstärke wäre dies z.B. „Ampere“, für die Länge „Meter“ usw. Die SI-Einheit muss für jede Größenart definiert werden. Der Verweis auf diese physikalische Einheit wird in diesem Attribut gespeichert.

m_strFormulaSymbol Datentype: QString
Einer physikalischen Größe wird in mathematischen Gleichungen ein Schriftzeichen zugeordnet, das man *Formelzeichen* nennt. Dieses ist grundsätzlich willkürlich, jedoch existieren eine Reihe von Konventionen (z.B. DIN 1304) zur Bezeichnung bestimmter Größen. Dieses Formelzeichen wird in diesem Attribut gespeichert und wird verwendet, um den „lesbaren“ String für die mathematischen Umrechnungsfunktionen der Einheiten zu erzeugen.

m_bIsPowerRelated Datentype: bool
Dieses Flag wird zur Ermittlung des Faktors für logarithmische Einheiten. Soll von Feldgrößen ausgehend ein Pegel oder Maß berechnet werden, steht dadurch das Verhältnis der Quadrate dieser Größen und es gilt
$$L = 10\lg\frac{P_2}{P_1}\,\mathrm{dB} = 10\lg\frac{\tilde{x}_2^2}{\tilde{x}_1^2}\,\mathrm{dB} = 20\lg\frac{\tilde{x}_2}{\tilde{x}_1}\,\mathrm{dB}$$
Für „PowerRelated“ Größen ist der Faktor = 20, für die anderen (Feldgrößen) ist der Faktro = 10.

m_bInitialized Datentype: bool
Die Größenarten und ihre Einheiten müssen durch den „*initialize*“ Aufruf der Klasse *CPhysSize* initialisiert werden. Durch den „*initialize*“ Aufruf wird das Flag „*Initialized*“ auf *true* gesetzt. Erst eine vollständig „initialisierte“ Größenart und Einheit ist in der Lage, Einheiten umzurechnen. Für weitere Informationen siehe *initialize* Methode der Klasse *CPhysSize*.

5.6.2. Konstruktoren und Destruktor

5.6.2.1. *CPhysSize(const QString, const QString&, const QString&, const QString&, const QString&, bool)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse. Der Konstruktor der Basisklasse *CUnitGrp* wird implizit so aufgerufen, dass eine Instanz vom Einheiten-Type „PhysUnit“ erzeugt wird.
Parameter:
i_strGroupName (IN) Datentype: const QString&
Name für das Wissensgebiet, dem die physikalische Größe zuzuordnen ist. Mögliche Werte sind „Electricity“, „Geometry“, „Kinematics“, …
i_strName (IN) Datentype: const QString&
Bezeichnet die Größenart. Mögliche Werte sind „Voltage“, „Current“, „Length“, „Frequency“, „Velocity“, …
i_strSIUnitName (IN) Datentype: const QString&
(Ausgeschriebener) Name der SI Einhet der Größenart. Mögliche Werte sind „Volt“, „Ampere“, „Meter“, „Hertz“, …
i_strSIUnitSymbol (IN) Datentype: const QString&
Symbol der SI Einhet der Größenart. Mögliche Werte sind „V“, „A“, „m“, „Hz“, …
i_strFormulaSymbol Datentype: QString
Einer physikalischen Größe wird in mathematischen Gleichungen ein Schriftzeichen zugeordnet, das man *Formelzeichen* nennt (siehe auch gleichnamiges Attribut der Klasse). Mögliche Werte sind „U“, „I“, „L“, „f“, …

5.6.2.2. *~CPhysSize ()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Destruktor. Zerstört die Instanz und trägt die Größenart dabei wieder aus dem Objekt-Pool aus.

5.6.3. Operationen

5.6.3.1. *operator ==*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Speicheradresse identisch ist.
Rückgabewert: .....Datentype: bool

5.6.3.2. *operator !=*

Besitzerbereich: Instanz

Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Speicheradresse identisch ist.
Rückgabewert: .....Datentype: bool

5.6.3.3. *initialize(bool)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Wurden alle Einheiten einer physikalischen Größenart erzeugt, muss die SI-Einheit ermittelt, müssen die Einheiten für die „*findBestUnit*“ Funktionalität verkettet, muss Speicher für die Tabelle interner Umrechnungsfunktionen reserviert und es müssen die internen Umrechnungsfunktionen ermittelt und in die Umrechnungstabelle eingetragen werden. All diese Initialisierungs-Aufgaben kann die Klasse *CPhysSize* durch die „*initialize*“ Operation automatisch erledigen, die innerhalb des Konstruktors der von *CPhysSize* abgeleiteten Klasse CPhysSize<PhysicalQuantity> aufgerufen werden sollte. Durch den „*initialize*“ Aufruf wird das Flag „*Initialized*“ auf *true* gesetzt. Erst eine vollständig „initialisierte“ Größenart und Einheit ist in der Lage, Einheiten umzurechnen.
Parameter:
i_bCreateFindBestChainedList (IN) Datentype: bool
Bei der automatischen Verkettung der Einheiten für die „*findBestUnit*“ Funktionalität bedient sich die Klasse *CPhysSize* eines recht einfachen Algorithmus, bei dem fortlaufend beginnend ab der ersten Einheit der Größenart die Einheiten über „*Prev*“ und „*Next*“ verkettet werden. Abgebrochen wird die Kette mit Erreichen der ersten logarithmischen Einheit. Damit dieser Algorithmus funktioniert, müssen die Einheiten beginnend mit der „niedrigsten“ Einheit der Größenart in aufsteigender Reihenfolge angelegt und als „*Childs*“ der Größenart in den Einheiten Objekt Pool eingetragen worden sein. Der Verkettungs-Automatismus innerhalb der *initialize* Methode kann über dieses Flag ausgeschaltet werden. In diesem Fall können die Einheiten der Größenart „manuell“ durch Aufruf der Methode „*setNextLowerHigherUntis*“ der Klasse *CUnit* verkettet werden. Die „eigenhändige“ Verkettung der Einheiten einer Größenart ist z.B. dann zu verwenden, wenn Einheiten wie „Inch“, „Feet“, „cm“ und „dm“ (also Einheiten, die nicht über einen durch 10^3 ganzzahlig teilbaren Faktor aus der SI Einheit abgeleitet werden können) aus der „*findBestUnit*“ Kette ausgeschlossen werden sollen.
Rückgabewert: .....Datentype: bool

5.6.3.4. *addFctConvert(CPhysSize&, CPhysSize&, EFctConvert)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Um eine Einheit in eine Einheit einer anderen Größenart umzurechnen, werden „externe“ Umrechnungsfunktionen verwendet, die zur Umrechnung eine Referenzeinheit benötigen. Der „normale“ Weg, eine Einheit in eine Einheit einer anderen Größenart umzurechnen ist, die Quell-Einheit zunächst in deren SI Einheit umzurechnen, anschließend den Wert in die SI Einheit der Zieleinheit umzurechnen und im dritten Schritt den Wert in die Zieleinheit zu konvertieren. Bei der Umrechnung einer logarithmischen Quell-Einheit in eine

logarithmische Zieleinheit sind diese drei Rechenschritte unnötig, da zur Umrechnung lediglich konstante Summanden zu addieren sind.
Über die „*addFctConvert*“ kann eine direkte Umrechnungsfunktion in eine andere Größenart unter Verwendung der definierten Referenzeinheit festgelegt werden. Allerdings nur, wenn eine der folgenden, mathematischen Operationen angewendet werden muss, um eine Einheit der Quellgrößenart in eine Einheit der Zielgrößenart umzurechnen:
„x*r“, „x²/r“, „x/r“, „√(x/r)“, „x²*r“, „√(x/r)“
Diese mathematischen Funktionen reichten bislang aus, um alle wirklich benötigten Umrechnungen zwischen unterschiedlichen Größenarten zu realisieren. Die Einschränkung gilt wegen der implementierten Automatismen, die bei der Konvertierung der Einheiten zum Tragen kommen, um die resultierende Umrechnungsfunktion mit den anzuwendenden Konstanten „m“, „t“, und „k“ aus den Umrechnungsfunktionen der Quell- und Ziel-Einheit dynamisch zu ermitteln.
Anmerkung:
Wie erwähnt, war es bislang nicht notwendig, „komplexere“ Umrechnungsfunktionen festlegen zu müssen. Sollte dies einmal notwendig sein, würde es sich anbieten, die Möglichkeit zu implementieren, benutzerdefinierte Umrechnungsfunktionen anlegen zu können.
Parameter:
i_physSizeDst (IN) Datentype: CPhysSize&
Größenart, in die die Quellgrößenart umzurechnen ist.
i_physSizeRef (IN) Datentype: CPhysSize&
Referenzeinheit, die bei der Umrechnung zu verwenden ist. Der Referenzwert wird von der Größenart der Referenzeinheit ausgelesen.
i_fctConvert (IN) Datentype: EFctConvert
Mathematische Umrechnungsfunktion, um die Quelleinheit in die Zieleinheit umzurechnen. Dabei gelten die oben beschriebenen Einschränkungen.

5.6.3.5. *getSIUnitName*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den (ausgeschriebenen) Name der SI Einheit der Größenart zurück.
Rückgabewert: .....Datentype: QString

5.6.3.6. *getSIUnitSymbol*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt das Einheiten-Symbol der SI Einheit der Größenart zurück.
Rückgabewert: .....Datentype: QString

5.6.3.7. *getSIUnit*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt einen Verweis auf die SI Einheit der Größenart zurück.

Rückgabewert: .....Datentype: CPhysUnit*

5.6.3.8. *getFormulaSymbol*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt das Formelzeichen der Größenart zurück (siehe Attribut *FormulaSymbol* der Klasse).
Rückgabewert: .....Datentype: QString

5.6.3.9. *isPowerRelated*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt an, ob es sich bei der Größenart um eine Feldgröße oder um eine Pegelmaß handelt (siehe Attribut *IsPowerRelated* der Klasse).
Rückgabewert: .....Datentype: bool

5.6.3.10. *getPhysUnitCount*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die Anzahl der physikalischen Einheiten zurück, die innerhalb der Größenart angelegt wurden.
Rückgabewert: .....Datentype: unsigned int

5.6.3.11. *getPhysUnit(unsigned int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Die Einheiten werden als „Childs“ der Größenart in einer Liste innerhalb des Unit Objekt Pools angelegt. Die Methode liefert die Einheit am entsprechenden Index in der Liste zurück.
Parameter:
i_udx (IN) Datentype: unsigned int
Index in die Liste der Einheiten der Größenart.
Rückgabewert: .....Datentype: CPhysUnit*

5.6.3.12. *findPhysUnit(const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Durchsucht die Liste der Einheiten der Größenart nach einer Einheit, deren Symbol oder Name dem übergebenen String entspricht.
Parameter:

i_strSymbolOrName (IN) Datentype: QString
Symbol oder (ausgeschriebener) Name der Einheit, für die eine Referenz benötigt wird.
Rückgabewert: ..... Datentype: CPhysUnit*

5.6.3.13. *findPhysUnitBySymbol(const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Durchsucht die Liste der Einheiten der Größenart nach einer Einheit, deren Symbol dem übergebenen String entspricht.
Parameter:
i_strSymbol (IN) Datentype: QString
Symbol der Einheit, für die eine Referenz benötigt wird.
Rückgabewert: ..... Datentype: CPhysUnit*

5.6.3.14. *findPhysUnitByName(const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Durchsucht die Liste der Einheiten der Größenart nach einer Einheit, deren Name dem übergebenen String entspricht.
Parameter:
i_strName (IN) Datentype: QString
(Ausgeschriebener) Name der Einheit, für die eine Referenz benötigt wird.
Rückgabewert: ..... Datentype: CPhysUnit*

5.6.3.15. *getRefVal(CPhysUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Virtuelle Methode die dann zu überschreiben ist, wenn die Größenart einen Referenzwert zur Verfügung stellen muss, um zwischen Einheiten unterschiedlicher Größenarten umzurechnen. Soll z.B. eine elektrische Leistung in eine elektrische Spannung konvertiert werden, wird hierzu der Referenzwiderstand benötigt und die Größenart „elektrischer Widerstand“ muss die Methode „getRefVal“ implementieren, um so z.B. den Referenzwiderstand „50 Ω„ zurückzugeben.
Parameter:
i_pPhysUnitRef (IN) Datentype: *CPhysUnit**
Verweis auf die Einheit, in der der Referenzwert benötigt wird. Wird hier der Wert NULL übergeben, wird der Referenzwert in der SI Einheit der Größenart erwartet.
Rückgabewert: ..... Datentype: double

5.7. Klasse(n) CPhysSize<PhysicalQuantity>

Wie bereits erwähnt werden nicht alle Größenarten sowie alle möglichen Maßeinheiten einer Größenart innerhalb ein und derselben Applikation verwendet werden. Deshalb sollte

für jede tatsächlich verwendete Größenart ein von *CPhysSize* abgeleitete Klasse implementiert werden. Innerhalb dieser Klasse sollten alle verwendeten Maßeinheiten als Member-Variablen angelegt und somit zusammen mit der Größenart erzeugt und wieder zerstört werden. Ggf. kann man hier bereits auf eine bereits implementierte und veröffentliche Klasse innerhalb einer Bibliothek zurückgreifen. Beispiele, wie eine solche, von CPhysSize abgeleitete Klasse, definiert und implementiert werden kann, finden sich innerhalb der ZSQtLib unterhalb des Verzeichnis „ZSPhysSizes“.

5.8. Klasse *CUnitRatio*

Instanzen dieser Klasse repräsentieren reine Quotienten- und Verhältnisgrößen, wie z.B. „PerCent“, „PerMille“, etc. Die Ratio-Instanzen sind bereits innerhalb der *ZSPhysVal* Bibliothek angelegt. Die Ratio-Instanzen spielen bei Multiplikationen und Divisionen von physikalischen Größenwerten sowie bei der Angabe der Ungenauigkeiten physikalischer Größenwerte eine besondere Stellung ein.

5.8.1. Attribute

m_fFactor Datentype: double
Faktor, mit dem der ursprüngliche Wert zu multiplizieren ist (z.B. 0,01 für PerCent).

5.8.2. Konstruktoren und Destruktor

5.8.2.1. *CUnitRatio(CUnitGrp&, const QString&, const QString&, double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse. Als Unit-Group ist ein Verweis auf die Größenart „Ratio“ zu übergeben.
Parameter:
i_unitGrp (IN) Datentype: const CUnitGrp&
Verweis auf die Größenart „Ratio“. Eigentlich ist dieser Parameter unnötig, da Ratio-Instanzen immer nur genau dieser Größenart angehören können und dürfen. Durch die Übergabe der Unit-Gruppe wurde die Implementierung eines zusätzliche *CUnit* Konstruktors umgangen.
i_strName (IN) Datentype: const QString&
(Ausgeschriebener) Name der Einheit, z.B. „PerCent“, „PerMille“, „PerOne“.
i_strSymbol (IN) Datentype: const QString&
Symbol der Verhältnisgröße, z.B. „%“, „‰“ oder ein Leerstring für „PerOne“.
i_fFactor (IN) Datentype: double
Faktor, mit dem der ursprüngliche Wert zu multiplizieren ist (z.B. 0,01 für PerCent).

5.8.2.2. *~CUnitRatio ()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Destruktor. Zerstört die Instanz und trägt die Größenart dabei wieder aus dem Objekt-Pool aus.

5.8.3. Operationen

5.8.3.1. *getFactor*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Faktor zurück, mit dem der ursprüngliche Wert zu multiplizieren ist (z.B. 0,01 für PerCent).
Rückgabewert: .....Datentype: double

5.8.3.2. *isConvertible(const CUnit&, double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Überschreibt die entsprechende virtuelle Methode der Basisklasse. Testet, ob die Einheit in die definierte Zieleinheit konvertiert werden kann.
Parameter:
i_unitDst (IN) Datentype: CUnit&
Zieleinheit, in die der Wert zu konvertieren wäre.
i_fVal (IN) Datentype: double
Wert, der zu konvertieren wäre (Default = 1.0)
Rückgabewert: .....Datentype: bool

5.8.3.3. *convertValue(double, const CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Überschreibt die entsprechende virtuelle Methode der Basisklasse. Konvertiert den übergebenen Wert aus der Einheit, für die die Methode aufgerufen wurde, in die definierte Zieleinheit. Beispiel:

```
double      fVal_PerCent = 10.0;
CUnitRatio& unitPerCent = Ratio().PerCent ();
CUnitRatio& unitPerMille = Ratio().PerMille();
double      fVal_PerMille = unitPerCent.convertValue(fVal_PerCent,unitPerMille);
Ergebnis:   fVal_PerMille = 1.0;
```

Parameter:
i_fVal (IN) Datentype: double
Zu konvertierender Wert in der Einheit, für die die Methode aufgerufen wird.
i_unitDst (IN) Datentype: CUnit&
Zieleinheit, in die der Wert zu konvertieren ist.
Rückgabewert: Datentype: double
Wert in der Zieleinheit.

5.9. Klasse *CUnitGrpRatio*

Klasse zur Verwaltung der Gruppe der Verhältnisgrößen „Prozent“, „Promille“ und „Eins“, die als Member-Elemente angelegt sind. Eine Instanz der Klasse wird durch Aufruf der *initDll* Methode des Subsystem ZSPhysVal erzeugt. Über die Methode „*Ratio*“ kann auf

diese Instanz zugegriffen werden. Der Zugriff auf die Ratio-Einheiten erfolgt über die Methoden *PerOne*, *PerCent* und *PerMille*. Beispiel:

```
CUnitRatio& unitRatioPerCent = Ratio().PerCent().
```

5.10. Klassen *CUnitSIBase* und *CUnitGrpSIBase*

Die Einheiten der sieben SI-Basisgrößen Meter, Kilogramm, Sekunde, Ampere, Kelvin, Mol und Candela sind als Member-Elemente der Klasse *CUnitGrpSIBase* angelegt. Eine Instanz dieser Einheitengruppe wird durch Aufruf der *initDll* Methode des Subsystem ZSPhysVal erzeugt. Über die Methode „*SIBase*“ kann auf diese Instanz zugegriffen werden. Die sieben SI-Basis-Einheiten sind als Member-Elemente der Klasse angelegt und können über die Methoden *Meter*, *Kilogram*, *Second*, *Ampere*, *Kelvin*, *Mol* und *Kandela* erreicht werden.

Weder die Gruppenklasse noch die SIBase Einheiten-Klasse sind von besonderer Bedeutung und werden eigentlich nicht verwendet. Sie wurden nur der Vollständigkeit halber implementiert.

5.11. Klasse *CPhysValRes*

Die Klasse *CPhysValRes* kapselt die den physikalischen Werten anbehaftete Ungenauigkeit. Die Klasse besitzt eine große Anzahl von Konstruktoren und *setVal* Methoden, um Instanzen aus double Werten und Strings, mit und ohne Einheit, als Messunsicherheit oder als Einstellgenauigkeit zu erzeugen.

Wie bereits erwähnt, sind Messwerte fehlerbehaftet und Einstellwerte können nur mit einer bestimmten Genauigkeit vorgenommen werden. Werden physikalische Werte addiert oder subtrahiert, summiert sich ihre Ungenauigkeit. Werden physikalische Werte multipliziert oder dividiert, muss ihre Ungenauigkeit multipliziert werden. Für Einstellwerte gilt dies jedoch nicht. Hier hängt es davon ab, was mit dem Wert im Endeffekt geschehen soll. Wird der resultierende Wert wieder als Einstellparamter verwendet, so ist die resultierende Auflösung die Auflösung des neuen Einstellparameters. Verwendet man den Einstellparameter zur Berechnung eines Messwerts, um diesen Wert anzuzeigen, müsste die Genauigkeit des Einstellparameters berücksichtigt werden. Diese wurde jedoch nicht im Klassenmodell berücksichtigt und dieser Anwendungsfall wird somit nicht generisch gelöst. Ist aber wohl auch nur ein imaginärer Anwendungsfall und (bislang zumindest) in der praktischen Anwendung ohne Bedeutung.

5.11.1. Attribute

m_resType Datentype: EResType
: Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt. Dieses Attribut ist bei der Bestimmung der Anzahl auszugebender, unsicherer Dezimalstellen zu berücksichtigen, aber auch bei mathematischen Operationen mit physikalischen Werten.

m_fVal Datentype: double
: Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an.

m_pUnitGrp Datentype: CUnitGrp*
: Verweis auf die Größenart der Messunsicherheit. Dies muss mit der Größenart des

physikalischen Wertes übereinstimmen, dessen Ungenauigkeit die *CPhysValRes* Instanz repräsentiert.

m_pUnit Datentype: CUnit*
Verweis auf die Einheit der Messunsicherheit. Dies muss derselben Größenart angehören wie auch die Einheit des physikalischen Wertes, dessen Ungenauigkeit die *CPhysValRes* Instanz repräsentiert.

m_strVal Datentype: QString
Wird die Ungenauigkeitsangabe in einen String konvertiert, wird das Ergebnis in dieser Instanzvariablen gespeichert, um nachfolgende *toString* Aufrufe zu beschleunigen. Wird die Ungenauigkeitsangabe verändert, wird der String invalidiert (auf einen Leerstring gesetzt).

5.11.2. Konstruktoren und Destruktor

5.11.2.1. *CPhysValRes(EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert jedoch noch nicht gültig ist.
Parameter:
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.11.2.2. *CPhysValRes(double, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert gültig ist.
Parameter:
i_fVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.11.2.3. *CPhysValRes(const CPhysValRes&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Copy Constructor.

5.11.2.4. *CPhysValRes(CUnitGrp&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:

Erzeugt eine Instanz der Klasse, deren Wert jedoch noch ungültig ist. Die Größenart wird bereits beim Anlegen der Instanz festgelegt.

Parameter:

i_unitGrp (IN) Datentype: CUnitGrp&
Referenz auf die Größenart der Ungenauigkeit.

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.11.2.5. *CPhysValRes(CUnit*, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt eine Instanz der Klasse, deren Wert jedoch noch ungültig ist. Die Einheit wird bereits beim Anlegen der Instanz festgelegt.

Parameter:

i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit der Ungenauigkeit (darf auch NULL sein).

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.11.2.6. *CPhysValRes(CUnit&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt eine Instanz der Klasse, deren Wert jedoch noch ungültig ist. Die Einheit wird bereits beim Anlegen der Instanz festgelegt.

Parameter:

i_unit (IN) Datentype: CUnit&
Referenz auf die Einheit der Ungenauigkeit (kann nicht NULL sein).

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.11.2.7. *CPhysValRes(CUnitRatio&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt die Instanz der Klasse, deren Wert als eine Verhältnisgröße zu interpretieren ist. Der Wert ist jedoch noch ungültig.

Parameter:

i_unitRatio (IN) Datentype: CUnitRatio&
Referenz auf eine Einheit der Größenart *Ratio*.

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.11.2.8. *CPhysValRes(CPhysUnit&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt die Instanz der Klasse, deren Wert als absoluter Wert zu interpretieren ist. Der Wert ist jedoch noch ungültig.

Parameter:
i_physUnit (IN) Datentype: CPhysUnit&
Referenz auf die physikalische Einheit.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.11.2.9. *CPhysValRes(double, CUnit*, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt eine Instanz der Klasse mit einem gültigen Wert. Die Einheit wird bereits beim Anlegen der Instanz festgelegt.

Parameter:
i_fVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an.
i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit der Ungenauigkeit (darf auch NULL sein).
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.11.2.10. *CPhysValRes(double, CUnitRatio&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt die Instanz der Klasse mit einem gültigen Wert. Der Wert ist als Verhältnisgröße zu interpretieren.

Parameter:
i_fVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an.
i_unitRatio (IN) Datentype: CUnitRatio&
Referenz auf eine Einheit der Größenart *Ratio*.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.11.2.11. *CPhysValRes(double, CPhysUnit&, EResType)*

Besitzerbereich: Instanz

Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt die Instanz der Klasse mit einem gültigen Wert. Der Wert ist als absoluter Wert zu interpretieren.

Parameter:

i_fVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an.

i_physUnit (IN) Datentype: CPhysUnit&
Referenz auf die physikalische Einheit.

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.11.2.12. *CPhysValRes(const QString&, CUnit*, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit enthalten kann. Falls der String die Angabe der Einheit nicht enthält, es aber bereits bekannt ist, welche Einheit zu verwenden ist, kann die Einheit an den Konstruktor mit übergeben werden. Enthält der String lediglich das Symbol der Einheit, muss die Größenart angegeben werden. Dies kann durch Übergabe der SI-Einheit der Größenart geschehen.
Beispiel:

```
CPhysValRes physValRes( “0.012 µs”, &Kinematics::Time().Second() );
```

Ergebnis:

```
physValRes.m_type = EResTypeResolution
physValRes.m_fVal = 0.012
physValRes.m_pUnitGrp = &Kinematics::Time()
physValRes.m_pUnit = &Kinematics::Time().MicroSecond()
```

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String, der den numerischen Wert enthält, der in eine Ungenauigkeitsangabe zu konvertieren ist. Der Wert „0“ zeigt einen ungültigen Wert an.

i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit der Ungenauigkeit (darf auch NULL sein).

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.11.2.13. *CPhysValRes(const QString&, CUnitRatio&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen der Größenart Ratio und deren Einheit enthalten kann.

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String, der den numerischen Wert enthält, der in eine Ungenauigkeitsangabe zu konvertieren ist. Der Wert „0“ zeigt einen ungültigen Wert an.
i_unitRatio (IN) Datentype: CUnitRatio&
Referenz auf eine Einheit der Größenart Ratio.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.
Beispiel:

```
CPhysValRes physValRes( “0.012 %”, Ratio().PerCent() );
```

Ergebnis:

```
physValRes.m_type = EResTypeResolution
physValRes.m_fVal = 0.012
physValRes.m_pUnitGrp = &Ratio()
physValRes.m_pUnit = &Ratio().PerMille()
```

5.11.2.14. *CPhysValRes(const QString&, CPhysUnit&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen des Wissensgebiets, der Größenart und deren Einheit enthalten kann. Beim Anlegen der Instanz muss bereits die Größenart des Werts bekannt sein.
Parameter:
i_strVal (IN) Datentype: const QString&
Referenz auf den String, der den numerischen Wert enthält, der in eine Ungenauigkeitsangabe zu konvertieren ist. Der Wert „0“ zeigt einen ungültigen Wert an.
i_physUnit (IN) Datentype: CPhysUnit&
Referenz auf die physikalische Einheit.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.
Beispiel:

```
CPhysValRes physValRes( “0.012 µs”, Kinematics::Time().Second() );
```

Ergebnis:

```
physValRes.m_type = EResTypeResolution
physValRes.m_fVal = 0.012
physValRes.m_pUnitGrp = &Kinemeatics::Time()
physValRes.m_pUnit = &Kinematics::Time().MicroSeconds()
```

5.11.2.15. *~ CPhysValRes()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Destruktor.

5.11.3. Operationen

5.11.3.1. *isValid*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Ungenauigkeitsangaben sind immer Werte größer als 0. Einer Instanz der Klasse muss also ein Wert zugewiesen worden sein, bevor sie verwendet werden kann.
Rückgabewert: .....Datentype: bool

5.11.3.2. *type*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt zurück, ob es sich bei der Instanz um eine Ungenauigkeitsangabe eines Messwerts oder um eine Auflösung eines Einstellparameters handelt.
Rückgabewert: .....Datentype: EResType

5.11.3.3. *setUnitGrp(CUnitGrp*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Legt die Größenart fest.
Parameter:
i_pUnitGrp (IN) Datentype: CUnitGrp*
Verweis auf die Größenart.

5.11.3.4. *getUnitGrp*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt einen Verweis auf die Größenart zurück.
Rückgabewert: .....Datentype: CUnitGrp*

5.11.3.5. *setUnit(CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Legt die Einheit fest.
Parameter:
i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit.

5.11.3.6. *getUnit*

Besitzerbereich: Instanz

Sichtbarkeit: öffentlich
Beschreibung:
Gibt einen Verweis auf die Einheit zurück.
Rückgabewert: .....Datentype: CUnit*

5.11.3.7. *setVal(double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt den Größenwert der Instanz.
Parameter:
i_fVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an.

5.11.3.8. *setVal(double, CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt den Größenwert der Instanz und dessen Einheit fest.
Parameter:
i_fVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an.
i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit der Ungenauigkeit (darf auch NULL sein).

5.11.3.9. *setVal(double, CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt den Größenwert der Instanz und dessen Einheit fest.
Parameter:
i_fVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an.
i_unit (IN) Datentype: CUnit*
Referenz auf die Einheit der Ungenauigkeit.

5.11.3.10. *setVal(const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Bildet den Größenwert der Instanz aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit enthalten kann. Falls der String die Angabe der Einheit nicht enthält, muss diese der Instanz zuvor bereits bekannt gegeben worden sein.
Beispiel 1:

```
CPhysValRes physValRes;
physValRes.setVal( “0.012 Kinematics::Time::µs” );
```

Ergebnis:

```
physValRes.m_type = EresTypeResolution;
physValRes.m_fVal = 0.012
physValRes.m_pUnitGrp = &Kinematics::Time()
physValRes.m_pUnit = &Kinematics::Time().MicroSecond()
```

Beispiel 2:

```
CPhysValRes physValRes( Kinematics::Time().MicroSeconds() );
physValRes.setVal( "0.012" );
```

Ergebnis:

```
physValRes.m_type = EresTypeResolution;
physValRes.m_fVal = 0.012
physValRes.m_pUnitGrp = &Kinematics::Time()
physValRes.m_pUnit = &Kinematics::Time().MicroSecond()
```

Parameter:

i_strVal (IN) Datentype: const QString&

Referenz auf den String, der den numerischen Wert enthält, der in eine Ungenauigkeitsangabe zu konvertieren ist. Der Wert „0“ zeigt einen ungültigen Wert an.

5.11.3.11. *setVal(const QString&, CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Bildet den Größenwert der Instanz aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit enthalten kann. Falls der String die Angabe der Einheit nicht enthält, kann diese an die Methode übergeben werden oder muss der Instanz zuvor bereits bekannt gegeben worden sein.

Parameter:

i_strVal (IN) Datentype: const QString&

Referenz auf den String, der den numerischen Wert enthält, der in eine Ungenauigkeitsangabe zu konvertieren ist. Der Wert „0“ zeigt einen ungültigen Wert an.

i_pUnit (IN) Datentype: CUnit*

Verweis auf die Einheit der Ungenauigkeit (kann auch NULL sein).

5.11.3.12. *setVal(const QString&, CUnitRatio&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Bildet eine Ungenauigkeitsangabe als Verhältnisgröße aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für die Verhältniseinheit enthalten kann.

Parameter:

i_strVal (IN) Datentype: const QString&

Referenz auf den String, der den numerischen Wert enthält, der in eine Ungenauigkeitsangabe zu konvertieren ist. Der Wert „0“ zeigt einen ungültigen Wert an.

i_unit (IN) Datentype: CUnitRatio&

Referenz auf eine Einheit der Größenart Ratio.

5.11.3.13. *setVal(const QString&, CPhysUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Bildet eine Ungenauigkeitsangabe als Absolutwert aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für die Einheit enthalten kann.
Parameter:
i_strVal (IN) Datentype: const QString&
Referenz auf den String, der den numerischen Wert enthält, der in eine Ungenauigkeitsangabe zu konvertieren ist. Der Wert „0“ zeigt einen ungültigen Wert an.
i_physUnit (IN) Datentype: CPhysUnit&
Referenz auf die Einheit der Ungenauigkeit.

5.11.3.14. *getVal*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Größenwert der Ungenauigkeitsangabe zurück.
Rückgabewert: ..... Datentype: double

5.11.3.15. *getVal(const CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Größenwert der Ungenauigkeitsangabe in der gewünschten Einheit zurück.
Parameter:
i_pUnit (IN) Datentype: CUnit*
Verweis auf die gewünschte Einheit der Ungenauigkeit (darf auch NULL sein, dann wird der Wert nicht konvertiert).
Rückgabewert: ..... Datentype: double

5.11.3.16. *getVal(const CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Größenwert der Ungenauigkeitsangabe in der gewünschten Einheit zurück.
Parameter:
i_unit (IN) Datentype: CUnit&
Referenz auf die gewünschte Einheit der Ungenauigkeit.
Rückgabewert: ..... Datentype: double

5.11.3.17. *toString(EUnitFind, int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Konvertiert die Ungenauigkeitsangabe in einen String unter Berücksichtigung der gewünschten Formatierungsanweisungen.
Parameter:
i_unitFind (IN) Datentype: EUnitFind
Legt fest, ob der Wert in seiner „bestmöglichen“ Einheit ausgegeben werden soll.
i_iSubStrVisibility (IN) Datentype: int
Über einzelne Bits wird festgelegt, welcher Substring erzeugt werden soll (siehe enum EPhysValSubStr).
Rückgabewert: .....Datentype: QString

5.11.3.18. *toString(CUnit&, int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Konvertiert die Ungenauigkeitsangabe in einen String mit der gewünschten Einheit unter Berücksichtigung der gewünschten Formatierungsanweisungen.
Parameter:
i_unit (IN) Datentype: CUnit&
Referenz auf die Einheit, in der der Wert auszugeben ist.
i_iSubStrVisibility (IN) Datentype: int
Über einzelne Bits wird festgelegt, welcher Substring erzeugt werden soll (siehe enum EPhysValSubStr).
Rückgabewert: .....Datentype: QString

5.11.3.19. *operator ==*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.11.3.20. *operator !=*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.11.3.21. *operator = (const CPhysValRes&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Zuweisungungsoperator.

Rückgabewert: .....Datentype: CPhysValRes&

5.11.3.22. *operator + (const CPhysValRes&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Addiert zwei Instanzen der Klasse. Die Werte werden aufsummiert, die Einheiten ensprechend konvertiert.
Rückgabewert: .....Datentype: CPhysValRes

5.11.3.23. *operator += (const CPhysValRes&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Addiert eine Ungenauigkeitsangabe und weist das Ergebnis der Instanz zu. Die Werte werden aufsummiert, die Einheiten ensprechend konvertiert.
Rückgabewert: .....Datentype: CPhysValRes&

5.11.3.24. *operator * (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Multipliziert die Ungenauigkeitsangabe mit einem double Wert.
Rückgabewert: .....Datentype: CPhysValRes

5.11.3.25. *operator *= (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Multipliziert die Ungenauigkeitsangabe mit einem double Wert und weist das Ergebnis der Intanz zu.
Rückgabewert: .....Datentype: CPhysValRes

5.12. Klasse *CPhysVal*

Physikalische Werte sind zusammengesetzt aus einer Einheit sowie einer Auflösung (Resolution) bzw. Messgenauigkeit (Accuracy) und durch die Klasse *CPhysVal* repräsentiert. Physikalische Werte können in andere physikalische Einheiten umgerechnet werden. Die Werte lassen sich außerdem unter Berücksichtigung ihrer Auflösung bzw. Genauigkeit in Strings konvertieren und werden dabei automatisch mit der korrekten Anzahl gültigen Stellen ausgegeben. Es können aber auch noch weitere Formatierungsanweisungen berücksichtigt werden. Ferner können auch Strings eingelesen und in eine Instanz der Klasse *CPhysVal* überführt werden.

Operatoren der Klasse *CPhysVal* ermöglichen es, Instanzen der Klasse zu addieren und zu subtrahieren, wobei automatisch die notwendigen Einheiten-Konvertierungen und eine Korrektur der Genauigkeit durch die Operatoren vorgenommen wird.

Berücksichtigt wird ferner, dass Messwerte

- ungültig sein können (wenn die Messung gerade gestartet wurde und der Messwert noch nicht vorliegt),
- außerhalb des darstellbaren Zahlbereichs liegen können,
- zu genau wiedergegeben werden müssen, weil in einer Darstellung ohne Exponenten Nullen vor dem Komman eingefügt werden müssen, um den Wert darzustellen,
- zu ungenau wiedergegeben werden müssen, weil nicht genügend Platz zur Verfügung steht, um alle gültigen Stellen darzustellen,
- als Zwischenwerte vorliegen, weil eine Messung sehr lange dauert, und man Zwischenergebnisse darstellen will.

5.12.1. Attribute

m_pUnitGrp Datentype: CUnitGrp*
Verweis auf die Größenart des physikalischen Werts.
m_pUnit Datentype: CUnit*
Verweis auf die Einheit des physikalischen Werts.
m_validity Datentype: EValidity
Gibt an, ob der Wert gültig ist.
m_fVal Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an.
m_physValRes Datentype: CPhysValRes
Ungenaugkeit des physikalischen Werts.

5.12.2. Konstruktoren und Destruktor

5.12.2.1. *CPhysVal(EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert jedoch noch nicht gültig ist.
Parameter:
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.2. *CPhysVal(CUnitGrp&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert jedoch noch ungültig ist. Die Größenart wird bereits beim Anlegen der Instanz festgelegt.

Parameter:
i_unitGrp (IN) Datentype: CUnitGrp&
Referenz auf die Größenart des phsikalischen Werts.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.3. *CPhysVal(CUnit*, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert jedoch noch ungültig ist. Die Einheit wird bereits beim Anlegen der Instanz festgelegt.
Parameter:
i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit des physikalischen Werts (darf auch NULL sein).
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.4. *CPhysVal(CUnit&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert jedoch noch ungültig ist. Die Einheit wird bereits beim Anlegen der Instanz festgelegt.
Parameter:
i_unit (IN) Datentype: CUnit&
Referenz auf die Einheit des physikalischen Werts (kann nicht NULL sein).
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.5. *CPhysVal(const CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Copy Constructor.

5.12.2.6. *CPhysVal(double, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert gültig ist.
Parameter:

i_fVal (IN) Datentype: double
Größenwert.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.7. *CPhysVal(double, double, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert gültig ist und eine Genauigkeitsangabe besitzt.
Parameter:
i_fVal (IN) Datentype: double
Größenwert.
i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.8. *CPhysVal(double, CUnit*, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert gültig ist und eine Einheit besitzt.
Parameter:
i_fVal (IN) Datentype: double
Größenwert.
i_pUnit (IN) Datentype: CUnit*
Einheit des Größenwerts (kann auch NULL sein).
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.9. *CPhysVal(double, CUnit*, double, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert gültig ist und der eine Einheit sowie eine Ungenauigkeit besitzt.
Parameter:
i_fVal (IN) Datentype: double
Größenwert.
i_pUnit (IN) Datentype: CUnit*
Einheit des Größenwerts (kann auch NULL sein).

i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit in derselben Einheit wie der Größenwert selbst. Der Wert „0.0“ zeigt einen ungültigen Wert an.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.10. *CPhysVal(double, CUnit*, double, CUnit*, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert gültig ist und der eine Einheit sowie eine Ungenauigkeit besitzt.
Parameter:
i_fVal (IN) Datentype: double
Größenwert.
i_pUnit (IN) Datentype: CUnit*
Einheit des Größenwerts (kann auch NULL sein).
i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an.
i_pUnitRes (IN) Datentype: CUnit*
Einheit der Ungenauigkeitsangabe (kann auch NULL sein, in diesem Fall wird die Einheit des Größenwerts verwendet).
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.11. *CPhysVal(double, CUnit*, const CPhysValRes&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert gültig ist und der eine Einheit sowie eine Ungenauigkeit besitzt.
Parameter:
i_fVal (IN) Datentype: double
Größenwert.
i_pUnit (IN) Datentype: CUnit*
Einheit des Größenwerts (kann auch NULL sein).
i_physValRes (IN) Datentype: const CPhysValRes&
Ungenauigkeitsangabe.

5.12.2.12. *CPhysVal(double, CUnitRatio&, double, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert eine Verhältnisgröße und gültig ist und der eine Ungenauigkeit besitzen kann.

Parameter:
i_fVal (IN) Datentype: double
Größenwert.
i_unitRatio (IN) Datentype: CUnitRatio&
Einheit des Größenwerts als Einheit der Größenart Ratio.
i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit, ebenfalls als Verhältniszahl. Der Wert „0.0“ zeigt einen ungültigen Wert an.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.13. *CPhysVal(double, CUnitRatio&, double, CUnitRatio&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert eine Verhältnisgröße und gültig ist und der eine Ungenauigkeit besitzen kann.
Parameter:
i_fVal (IN) Datentype: double
Größenwert.
i_unitRatioVal (IN) Datentype: CUnitRatio&
Einheit des Größenwerts als Einheit der Größenart Ratio.
i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit, ebenfalls als Verhältniszahl. Der Wert „0.0“ zeigt einen ungültigen Wert an.
i_unitRatioRes (IN) Datentype: CUnitRatio&
Einheit der Ungenauigkeit als Einheit der Größenart Ratio.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.14. *CPhysVal(double, CUnitRatio&, const CPhysValRes&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert eine Verhältnisgröße und gültig ist und der eine Ungenauigkeit besitzt.
Parameter:
i_fVal (IN) Datentype: double
Größenwert.
i_unitRatioVal (IN) Datentype: CUnitRatio&
Einheit des Größenwerts als Einheit der Größenart Ratio.
i_physValRes (IN) Datentype: const CPhysValRes&
Ungenauigkeitsangabe.

5.12.2.15. *CPhysVal(double, CPhysUnit&, double, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert eine physikalische Größe und gültig ist und der eine Ungenauigkeit besitzen kann.

Parameter:
i_fVal (IN) Datentype: double
Größenwert.
i_physUnit (IN) Datentype: CPhysUnit&
Einheit des Größenwerts.
i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit als absoluter Wert. Der Wert „0.0“ zeigt einen ungültigen Wert an.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.16. *CPhysVal(double, CPhysUnit&, double, CUnitRatio&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert eine physikalische Größe und gültig ist und der eine Ungenauigkeit besitzen kann, die als Verhältnisgröße definiert ist.

Parameter:
i_fVal (IN) Datentype: double
Größenwert.
i_physUnitVal (IN) Datentype: CPhysUnit&
Einheit des Größenwerts.
i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit als absoluter Wert. Der Wert „0.0“ zeigt einen ungültigen Wert an.
i_unitRatioRes (IN) Datentype: CUnitRatio&
Einheit der Ungenauigkeit als Einheit der Größenart Ratio.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.17. *CPhysVal(double, CPhysUnit&, double, CPhysUnit&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert eine physikalische Größe und gültig ist und der eine Ungenauigkeit besitzen kann.

Parameter:
i_fVal (IN) Datentype: double
Größenwert.

i_physUnitVal (IN) Datentype: CPhysUnit&
Einheit des Größenwerts.
i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit als absoluter Wert. Der Wert „0.0“ zeigt einen ungültigen Wert an.
i_physUnitRes (IN) Datentype: CPhysUnit&
Einheit der Ungenauigkeit.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.18. *CPhysVal(double, CPhysUnit&, const CPhysValRes&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Wert eine physikalische Größe und gültig ist und der eine Ungenauigkeit besitzt.
Parameter:
i_fVal (IN) Datentype: double
Größenwert.
i_physUnitVal (IN) Datentype: CPhysUnit&
Einheit des Größenwerts.
i_physValRes (IN) Datentype: const CPhysValRes&
Ungenauigkeitsangabe.

5.12.2.19. *CPhysVal(const QString&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Falls der String die Angabe der Einheit enthält, muss ebenso das Wissengebiet als auch die Größenart innerhalb des Strings angegeben werden, da das Einheitensymbol nicht immer eindeutig ist und deshalb darauf verzichtet wurde, den gesamten Einheiten-Objekt-Baum nach dem Einheitensymbol bzw. dem Einheitennamen zu durchsuchen. Eine Ungenauigkeitsangabe muss durch das „±“ Symbol eingeleitet werden. Gelten die Einheitenangaben im String sowohl für den Wert als auch seine Ungenauigkeit, sind die beiden Werte in Klammern zu setzen und die Einheit nach dem in Klammern gesetzten Ausdruck einzufügen.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( “0.012” )
CPhysVal( “0.012 Kinematics::Time::ms” )
CPhysVal( “0.012 Kinematics::Time::MilliSeconds” )
CPhysVal( “0.012 Ratio::%” )
CPhysVal( “0.012 Ratio::PerCent” )
CPhysVal( “0.012 ± 0.001” )
CPhysVal( “0.012 Kinematics::Time::ms ± 1.0 Kinematics::Time::µs” )
CPhysVal( “(0.012 ± 0.001) Kinematics::Time::ms” )
CPhysVal( “0.012 Kinematics::Time::ms ± 1.0 Ratio::%” )
```

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.20. *CPhysVal(const QString&, double, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Falls der String die Angabe der Einheit enthält, muss ebenso das Wissengebiet als auch die Größenart innerhalb des Strings angegeben werden, da das Einheitensymbol nicht immer eindeutig ist und deshalb darauf verzichtet wurde, den gesamten Einheiten-Objekt-Baum nach dem Einheitensymbol bzw. dem Einheitennamen zu durchsuchen. Eine Ungenauigkeitsangabe muss durch das „±“ Symbol eingeleitet werden. Gelten die Einheitenangaben im String sowohl für den Wert als auch seine Ungenauigkeit, sind die beiden Werte in Klammern zu setzen und die Einheit nach dem in Klammern gesetzten Ausdruck einzufügen. Ist im String keine Ungenauigkeit mit angegeben, wird der an den Konstruktor übergebene Wert verwendet. Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( “0.012”, 0.001 )
CPhysVal( “0.012 Kinematics::Time::ms”, 0.001 )
CPhysVal( “0.012 Kinematics::Time::MilliSeconds”, 0.001 )
CPhysVal( “0.012 Ratio::%”, 0.001 )
CPhysVal( “0.012 Ratio::PerCent”, 0.001 )
CPhysVal( “0.012 Kinematics::Time::ms”, 0.001 )
CPhysVal( “(0.012 ± 0.001) Kinematics::Time::ms”, 0.001 )
CPhysVal( “0.012 Kinematics::Time::ms ± 1.0 Ratio::%”, 0.001 )
```

Parameter:
i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.
i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an. Wird ggf. durch eine Ungenauigkeitsangabe im String überschrieben.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.21. *CPhysVal(const QString&, CUnit*, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Wird eine gültige Einheiteninstanz mit an den Konstruktor übergeben, muss der String nicht unbedingt einen Einheitenstring enthalten. Ebenso muss auch weder das Wissensgebiet noch die Größenart im String

angegeben werden. Eine Ungenauigkeitsangabe muss durch das „±“ Symbol eingeleitet werden. Gelten die Einheitenangaben im String sowohl für den Wert als auch seine Ungenauigkeit, sind die beiden Werte in Klammern zu setzen und die Einheit nach dem in Klammern gesetzten Ausdruck einzufügen.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( “0.012”, Kinematics::Time().MilliSeconds() )
CPhysVal( “0.012 ms”, Kinematics::Time().Seconds )
CPhysVal( “(0.012 ± 0.001) ms”, Kinematics::Time().Seconds() )
CPhysVal( “0.012 ms ± 1.0 Ratio::%”, Kinematics::Time().Seconds() )
```

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.

i_pUnit (IN) Datentype: CUnit*
Einheit des Größenwerts (kann auch NULL sein). Wird ggf. durch die Angabe einer Einheit im zu konvertierenden String überschrieben. Wird ferner ggf. auch für die Ungenauigkeitsangabe verwendet, wenn hierfür nich explizit die Einheit im String angegeben wurde.

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.22. *CPhysVal(const QString&, CUnit*, double, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Wird eine gültige Einheiteninstanz mit an den Konstruktor übergeben, muss der String nicht unbedingt einen Einheitenstring enthalten. Ebenso muss auch weder das Wissensgebiet noch die Größenart im String angegeben werden. Eine Ungenauigkeitsangabe muss durch das „±“ Symbol eingeleitet werden. Gelten die Einheitenangaben im String sowohl für den Wert als auch seine Ungenauigkeit, sind die beiden Werte in Klammern zu setzen und die Einheit nach dem in Klammern gesetzten Ausdruck einzufügen. Ist im String keine Ungenauigkeit mit angegeben, wird der an den Konstruktor übergebene Wert verwendet.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( “0.012”, Kinematics::Time().MilliSeconds(), 0.001 )
CPhysVal( “0.012 ms”, Kinematics::Time().Seconds, 0.001 )
CPhysVal( “(0.012 ± 0.001) ms”, Kinematics::Time().Seconds(), 0.001 )
```

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.

i_pUnit (IN) Datentype: CUnit*
Einheit des Größenwerts (kann auch NULL sein). Wird ggf. durch die Angabe einer Einheit im zu konvertierenden String überschrieben. Wird ferner ggf. auch für die Ungenauigkeitsangabe verwendet, wenn hierfür nich explizit die Einheit im String angegeben wurde.

i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an. Wird ggf. durch eine Ungenauigkeitsangabe im String überschrieben.

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.23. *CPhysVal(const QString&, CUnit*, double, CUnit*, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Wird eine gültige Einheiteninstanz mit an den Konstruktor übergeben, muss der String nicht unbedingt einen Einheitenstring enthalten. Ebenso muss auch weder das Wissensgebiet noch die Größenart im String angegeben werden. Eine Ungenauigkeitsangabe muss durch das „±“ Symbol eingeleitet werden. Gelten die Einheitenangaben im String sowohl für den Wert als auch seine Ungenauigkeit, sind die beiden Werte in Klammern zu setzen und die Einheit nach dem in Klammern gesetzten Ausdruck einzufügen. Ist im String keine Ungenauigkeit mit angegeben, wird der an den Konstruktor übergebene Wert verwendet.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( “0.012”, Kinematics::Time().MilliSeconds(),
           0.001, Kinematics::Time().MilliSeconds() )
CPhysVal( “0.012 ms”, Kinematics::Time().Seconds,
          0.001, Kinematics::Time().MilliSeconds() )
CPhysVal( “0.012 ± 1.0”, Kinematics::Time().Seconds(),
          1.0, Kinematics::Time().MicroSeconds() )
```

Parameter:
i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.
i_pUnit (IN) Datentype: CUnit*
Einheit des Größenwerts (kann auch NULL sein). Wird ggf. durch die Angabe einer Einheit im zu konvertierenden String überschrieben.
i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an. Wird ggf. durch eine Ungenauigkeitsangabe im String überschrieben.
i_pUnitRes (IN) Datentype: CUnit*
Einheit der Ungenauigkeitsangabe (kann auch NULL sein). Wird ggf. durch die Angabe einer Einheit im zu konvertierenden String überschrieben.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.24. *CPhysVal(const QString&, CUnitGrp&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Größenart übergeben wird, muss der Einheitenstring nicht unbedingt auch die Angabe des

Wissensgebiets und der Größenart enthalten, sondern es reicht die Angabe des Einheitensymbols oder des (ausgeschriebenen) Einheitennamens. Eine Ungenauigkeitsangabe muss durch das „±“ Symbol eingeleitet werden. Gelten die Einheitenangaben im String sowohl für den Wert als auch seine Ungenauigkeit, sind die beiden Werte in Klammern zu setzen und die Einheit nach dem in Klammern gesetzten Ausdruck einzufügen.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( "0.012 ms", Kinematics::Time() )
CPhysVal( "0.012 ms ± 1.0 µs", Kinematics::Time() )
```

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.

i_unitGrp (IN) Datentype: CUnitGrp&
Größenart des Größenwerts. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.25. *CPhysVal(const QString&, CUnitGrp&, double, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Größenart übergeben wird, muss der Einheitenstring nicht unbedingt auch die Angabe des Wissensgebiets und der Größenart enthalten, sondern es reicht die Angabe des Einheitensymbols oder des (ausgeschriebenen) Einheitennamens. Eine Ungenauigkeitsangabe muss durch das „±“ Symbol eingeleitet werden. Gelten die Einheitenangaben im String sowohl für den Wert als auch seine Ungenauigkeit, sind die beiden Werte in Klammern zu setzen und die Einheit nach dem in Klammern gesetzten Ausdruck einzufügen.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( "0.012 ms", Kinematics::Time(), 0.001 )
```

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.

i_unitGrp (IN) Datentype: CUnitGrp&
Größenart des Größenwerts. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.

i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an. Wird ggf. durch eine Ungenauigkeitsangabe im String überschrieben.

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.26. *CPhysVal(const QString&, CUnitGrp&, CUnitGrp&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Größenart übergeben wird, muss der Einheitenstring nicht unbedingt auch die Angabe des Wissensgebiets und der Größenart enthalten, sondern es reicht die Angabe des Einheitensymbols oder des (ausgeschriebenen) Einheitennamens. Eine Ungenauigkeitsangabe muss durch das „±“ Symbol eingeleitet werden. Gelten die Einheitenangaben im String sowohl für den Wert als auch seine Ungenauigkeit, sind die beiden Werte in Klammern zu setzen und die Einheit nach dem in Klammern gesetzten Ausdruck einzufügen.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( "0.012 ms ± 1.0 %", Kinematics::Time(), Ratio() )
```

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.

i_unitGrpVal (IN) Datentype: CUnitGrp&
Größenart des Größenwerts. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.

i_unitGrpRes (IN) Datentype: CUnitGrp&
Größenart der Ungenauigkeit. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.27. *CPhysVal(const QString&, CUnitGrp&, double, CUnitRatio&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Größenart übergeben wird, muss der Einheitenstring nicht unbedingt auch die Angabe des Wissensgebiets und der Größenart enthalten, sondern es reicht die Angabe des Einheitensymbols oder des (ausgeschriebenen) Einheitennamens. Fehlt die Ungenauigkeitsangabe im String, werden die an den Konstruktor als Verhältniswert übergebenen Werten verwendet.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( "0.012 ms", Kinematics::Time(), 1.0, Ratio().PerCent() )
```

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.

i_unitGrpVal (IN) Datentype: CUnitGrp&
Größenart des Größenwerts. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.

i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an. Wird ggf. durch eine Ungenauigkeitsangabe im String überschrieben.
i_unitRatioRes (IN) Datentype: CUnitRatio&
Einheit der Größenart Ratio. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.28. *CPhysVal(const QString&, CUnitGrp&, double, CPhysUnit&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Größenart übergeben wird, muss der Einheitenstring nicht unbedingt auch die Angabe des Wissensgebiets und der Größenart enthalten, sondern es reicht die Angabe des Einheitensymbols oder des (ausgeschriebenen) Einheitennamens. Fehlt die Ungenauigkeitsangabe im String, werden die an den Konstruktor übergebenen Werte verwendet.
Beispiele für gültige Konstruktur-Aufrufe sind:
```
CPhysVal( “0.012 ms”, Kinematics::Time(), 1.0, Ratio().PerCent() )
```
Parameter:
i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.
i_unitGrpVal (IN) Datentype: CUnitGrp&
Größenart des Größenwerts. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.
i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an. Wird ggf. durch eine Ungenauigkeitsangabe im String überschrieben.
i_physUnit (IN) Datentype: CUnitRatio&
Physikalische Einheit. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.29. *CPhysVal(const QString&, CUnitGrp&, const CPhysValRes&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Größenart übergeben wird, muss der Einheitenstring nicht unbedingt auch die Angabe des

Wissensgebiets und der Größenart enthalten, sondern es reicht die Angabe des Einheitensymbols oder des (ausgeschriebenen) Einheitennamens. Fehlt die Ungenauigkeitsangabe im String, wird die an den Konstruktor übergebene Ungenauigkeitsangabe verwendet.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( "0.012 ms", Kinematics::Time(),
          CPhysValRes(0.001,Kinematics::Time().MilliSeconds()) )
```

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.

i_unitGrpVal (IN) Datentype: CUnitGrp&
Größenart des Größenwerts. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.

i_physValRes (IN) Datentype: const CPhysValRes&
Ungenauigkeitsangabe.

5.12.2.30. *CPhysVal(const QString&, CUnitRatio&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Einheit übergeben wird, muss der zu konvertierende String keine Einheitenangabe enthalten. Eine Ungenauigkeitsangabe muss durch das „±“ Symbol eingeleitet werden. Gelten die Einheitenangaben im String sowohl für den Wert als auch seine Ungenauigkeit, sind die beiden Werte in Klammern zu setzen und die Einheit nach dem in Klammern gesetzten Ausdruck einzufügen.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( "0.012", Ratio().PerOne() )
CPhysVal( "1.2 %", Ratio().PerOne() )
CPhysVal( "(0.012 ± 0.001) %", Ratio().PerOne())
```

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.

i_unitRatio (IN) Datentype: CUnitRatio&
Einheit der Größenart Ratio. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.31. *CPhysVal(const QString&, CUnitRatio&, double, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie

(optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Einheit übergeben wird, muss der zu konvertierende String keine Einheitenangabe enthalten.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( "0.012", Ratio().PerOne(), 0.00012 )
CPhysVal( "1.2 %", Ratio().PerOne(), 0.00012 )
```

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.

i_unitRatio (IN) Datentype: CUnitRatio&
Einheit der Größenart Ratio. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.

i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an. Wird ggf. durch eine Ungenauigkeitsangabe im String überschrieben.

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.32. *CPhysVal(const QString&, CUnitRatio&, double, CUnitRatio&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Einheit übergeben wird, muss der zu konvertierende String keine Einheitenangabe enthalten. Enthält der String nicht explizit eine Ungenauigkeitsangabe, werden die an den Konstruktor übergebenen Werte verwendet.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( "0.012", Ratio().PerOne(), 0.012, Ratio().PerCent() )
CPhysVal( "1.2 %", Ratio().PerOne(), 0.012, Ratio().PerCent() )
```

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.

i_unitRatioVal (IN) Datentype: CUnitRatio&
Einheit der Größenart Ratio. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.

i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an. Wird ggf. durch eine Ungenauigkeitsangabe im String überschrieben.

i_unitRatioRes (IN) Datentype: CUnitRatio&
Einheit der Größenart Ratio. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.33. *CPhysVal(const QString&, CUnitRatio&, const CPhysValRes&)*

Besitzerbereich: Instanz

Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Einheit übergeben wird, muss der zu konvertierende String keine Einheitenangabe enthalten. Enthält der String nicht explizit eine Ungenauigkeitsangabe, werden die an den Konstruktor übergebenen Werte verwendet.

Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( "0.012", Ratio().PerOne(),
          CPhysValRes(0.012,Ratio().PerCent()) )
```

Parameter:

i_strVal (IN) Datentype: const QString&

Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.

i_unitRatioVal (IN) Datentype: CUnitRatio&

Einheit der Größenart Ratio. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.

i_physValRes (IN) Datentype: const CPhysValRes&

Ungenauigkeitsangabe.

5.12.2.34. *CPhysVal(const QString&, CPhysUnit&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Einheit übergeben wird, muss der zu konvertierende String keine Einheitenangabe enthalten. Eine Ungenauigkeitsangabe muss durch das „±“ Symbol eingeleitet werden. Gelten die Einheitenangaben im String sowohl für den Wert als auch seine Ungenauigkeit, sind die beiden Werte in Klammern zu setzen und die Einheit nach dem in Klammern gesetzten Ausdruck einzufügen.

Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( "0.012", Kinematics::Time().MilliSeconds() )
CPhysVal( "1.2 µs", Kinematics::Time().Seconds() )
CPhysVal( "0.012 ± 0.001", Kinematics::Time().MilliSeconds() )
```

Parameter:

i_strVal (IN) Datentype: const QString&

Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.

i_physUnit (IN) Datentype: CPhysUnit&

Physikalische Einheit. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)

Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.35. *CPhysVal(const QString&, CPhysUnit&, double, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Einheit übergeben wird, muss der zu konvertierende String keine Einheitenangabe enthalten. Eine Ungenauigkeitsangabe muss durch das „±“ Symbol eingeleitet werden. Gelten die Einheitenangaben im String sowohl für den Wert als auch seine Ungenauigkeit, sind die beiden Werte in Klammern zu setzen und die Einheit nach dem in Klammern gesetzten Ausdruck einzufügen. Fehlt die Ungenauigkeitsangabe im String, wird der übergebene Wert verwendet.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( “0.012”, Kinematics::Time().MilliSeconds(), 0.001 )
CPhysVal( “1.2 µs”, Kinematics::Time().Seconds(), 0.001 )
CPhysVal( “0.012 ± 0.001”, Kinematics::Time().MilliSeconds(), 0.001 )
```

Parameter:
i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.
i_physUnit (IN) Datentype: CPhysUnit&
Physikalische Einheit. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.
i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an. Wird ggf. durch eine Ungenauigkeitsangabe im String überschrieben.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.36. *CPhysVal(const QString&, CPhysUnit&, double, CUnitRatio&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Einheit übergeben wird, muss der zu konvertierende String keine Einheitenangabe enthalten. Eine Ungenauigkeitsangabe muss durch das „±“ Symbol eingeleitet werden. Gelten die Einheitenangaben im String sowohl für den Wert als auch seine Ungenauigkeit, sind die beiden Werte in Klammern zu setzen und die Einheit nach dem in Klammern gesetzten Ausdruck einzufügen. Fehlt die Ungenauigkeitsangabe im String, wird der übergebene Wert verwendet und eine Ungenauigkeitsangabe als Verhältnisgröße gebildet.
Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( “0.012”, Kinematics::Time().MilliSeconds(),
          1.0, Ratio().PerCent() )
```

Parameter:
i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.
i_physUnitVal (IN) Datentype: CPhysUnit&
Physikalische Einheit. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.

i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an. Wird ggf. durch eine Ungenauigkeitsangabe im String überschrieben.
i_unitRatioRes (IN) Datentype: CUnitRatio&
Einheit der Größenart Ratio. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.37. *CPhysVal(const QString&, CPhysUnit&, double, CPhysUnit&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Einheit übergeben wird, muss der zu konvertierende String keine Einheitenangabe enthalten. Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( “0.012”, Kinematics::Time().MilliSeconds(),
          1.0, Kinematics::Time().MicroSeconds() )
```

Parameter:
i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.
i_physUnitVal (IN) Datentype: CPhysUnit&
Physikalische Einheit. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.
i_fResVal (IN) Datentype: double
Größenwert der Ungenauigkeit. Der Wert „0.0“ zeigt einen ungültigen Wert an. Wird ggf. durch eine Ungenauigkeitsangabe im String überschrieben.
i_physUnitRes (IN) Datentype: CPhysUnit&
Physikalische Einheit der Ungenauigkeit. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.12.2.38. *CPhysVal(const QString&, CPhysUnit&, const CPhysValRes&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse aus einem String, der einen numerischen Wert sowie (optional) Symbole und/oder Namen für Wissensgebiet, Größenart und Einheit sowie (optional) eine Ungenauigkeitsangabe enthalten kann. Da eine Referenz auf eine Einheit übergeben wird, muss der zu konvertierende String keine Einheitenangabe enthalten. Beispiele für gültige Konstruktur-Aufrufe sind:

```
CPhysVal( “0.012”, Kinematics::Time().MilliSeconds(),
          1.0, Kinematics::Time().MicroSeconds() )
```

Parameter:

i_strVal (IN) Datentype: const QString&
Referenz auf den String der in einen physikalischen Wert zu konvertieren ist.
i_physUnitVal (IN) Datentype: CPhysUnit&
Physikalische Einheit. Wird ggf. durch die Angabe eines Einheitenstrings im zu konvertierenden String überschrieben.
i_physValRes (IN) Datentype: const CPhysValRes&
Ungenauigkeitsangabe.

5.12.2.39. ~ *CPhysVal()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Destruktor.

5.12.3. Operationen

5.12.3.1. *isValid*

Besitzerbereich: Klasse
Sichtbarkeit: öffentlich
Beschreibung:
Solange der Instanz kein Wert zugewiesen wurde, ist sie Invalid.
Rückgabewert: .....Datentype: bool

5.12.3.2. *setValidity(EValidity)*

Besitzerbereich: Klasse
Sichtbarkeit: öffentlich
Beschreibung:
Weist der Instanz die übergebenen Validity-Flags zu.
Parameter:
i_validity (IN) Datentype: EValidity

5.12.3.3. *getValidity*

Besitzerbereich: Klasse
Sichtbarkeit: öffentlich
Beschreibung:
Gibt das Validity Flag der Instanz zurück.
Rückgabewert: .....Datentype: EValidity

5.12.3.4. *setUnitGrp(CUnitGrp*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Legt die Größenart fest.
Parameter:

i_pUnitGrp (IN) Datentype: CUnitGrp*
Verweis auf die Größenart.

5.12.3.5. *getUnitGrp*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt einen Verweis auf die Größenart zurück.
Rückgabewert: .....Datentype: CUnitGrp*

5.12.3.6. *setUnit(CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Legt die Einheit fest.
Parameter:
i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit.

5.12.3.7. *getUnit*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt einen Verweis auf die Einheit zurück.
Rückgabewert: .....Datentype: CUnit*

5.12.3.8. *setVal(double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt den Größenwert der Instanz. Nach Aufruf dieser Methode ist der Wert Valid.
Parameter:
i_fVal (IN) Datentype: double
Größenwert.

5.12.3.9. *setVal(double, CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt den Größenwert der Instanz und dessen Einheit fest. Nach Aufruf dieser Methode ist der Wert Valid.
Parameter:
i_fVal (IN) Datentype: double
Größenwert.

i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit (darf auch NULL sein).

5.12.3.10. *setVal(double, CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt den Größenwert der Instanz und dessen Einheit fest. Nach Aufruf dieser Methode ist der Wert Valid.
Parameter:
i_fVal (IN) Datentype: double
Größenwert.
i_unit (IN) Datentype: CUnit&
Referenz auf die Einheit

5.12.3.11. *setVal(const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Der übergebene String wird in den physikalischen Wert konvertiert. Für den String gelten dieselben Bedingungen, wie für den Konstruktor mit einem String als Eingabeparameter.
Parameter:
i_strVal (IN) Datentype: const QString&
String, der den zu konvertierenden Wert enthält.

5.12.3.12. *setVal(const QString&, CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Der übergebene String wird in den physikalischen Wert konvertiert. Für den String gelten dieselben Bedingungen, wie für den Konstruktor mit einem String und einer Einheit als Eingabeparameter.
Parameter:
i_strVal (IN) Datentype: const QString&
String, der den zu konvertierenden Wert enthält.
i_pUnit (IN) Datentype: CUnit*
Einheit, die zu verwenden ist, falls im String keine Einheitenangabe enthalten ist (darf auch NULL sein).

5.12.3.13. *setVal(const QString&, CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Der übergebene String wird in den physikalischen Wert konvertiert. Für den String gelten dieselben Bedingungen, wie für den Konstruktor mit einem String und einer Einheit als Eingabeparameter.

Parameter:
i_strVal (IN) Datentype: const QString&
String, der den zu konvertierenden Wert enthält.
i_unit (IN) Datentype: CUnit*
Referenz auf die Einheit, die zu verwenden ist, falls im String keine Einheitenangabe enthalten ist.

5.12.3.14. *getVal()*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Größenwert zurück. Das Validity Flag bleibt dabei unberücksichtig.
Rückgabewert: .....Datentype: double

5.12.3.15. *getVal(const CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Größenwert in der gewünschten Einheit zurück. Das Validity Flag bleibt dabei unberücksichtig.
Parameter:
i_strVal (IN) Datentype: const CUnit*
Gewünschte Einheit (darf auch NULL sein).
Rückgabewert: .....Datentype: double

5.12.3.16. *getVal(const CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Größenwert in der gewünschten Einheit zurück. Das Validity Flag bleibt dabei unberücksichtig.
Parameter:
i_strVal (IN) Datentype: const CUnit&
Referenz auf die gewünschte Einheit.
Rückgabewert: .....Datentype: double

5.12.3.17. *hasRes*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt true zurück, falls der physikalische Wert mit einer Ungenauigkeit behaftet ist.
Rückgabewert: .....Datentype: bool

5.12.3.18. *getRes*

Besitzerbereich: Instanz

Sichtbarkeit: öffentlich
Beschreibung:
Gibt die Ungenauigkeit der Instanz zurück.
Rückgabewert: ..... Datentype: CPhysValRes

5.12.3.19. *setRes(const CPhysValRes&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt die Ungenauigkeit der Instanz fest.
Parameter:
i_physValRes (IN) Datentype: const CPhysValRes&
Ungenauigkeit.

5.12.3.20. *toString(EUnitFind, int, EUnitFind, int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Konvertiert den physikalischen Wert in einen String unter Berücksichtigung der Ungenauigkeit sowie der spezifizierten Format-Anweisungen.
Parameter:
i_unitFindVal (IN) Datentype: EUnitFind (Default = FindNone)
Definiert, ob der Wert in der bestmöglichen Einheit auszugeben ist.
i_iValSubStrVisiblility (IN) Datentype: int (Default = SubStrVal | SubStrUnitSymbol)
Verodertes Bit-Feld über das festgelegt wird, welche Teilstrings für den Größenwert auszugeben sind (siehe enum EPhysValSubStr).
i_unitFindRes (IN) Datentype: EUnitFind (Default = FindNone)
Definiert, ob die Ungenauigkeit in der bestmöglichen Einheit auszugeben ist.
i_iResSubStrVisiblility (IN) Datentype: int (Default = SubStrNone)
Verodertes Bit-Feld über das festgelegt wird, welche Teilstrings die Angabe der Ungenauigkeit enthalten soll (siehe enum EPhysValSubStr).
Rückgabewert: ..... Datentype: QString

5.12.3.21. *toString(EUnitFind, int, CUnit&, int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Konvertiert den physikalischen Wert in einen String unter Berücksichtigung der Ungenauigkeit sowie der spezifizierten Format-Anweisungen.
Parameter:
i_unitFindVal (IN) Datentype: EUnitFind
Definiert, ob der Wert in der bestmöglichen Einheit auszugeben ist.
i_iValSubStrVisiblility (IN) Datentype: int
Verodertes Bit-Feld über das festgelegt wird, welche Teilstrings für den Größenwert auszugeben sind (siehe enum EPhysValSubStr).
i_unitRes (IN) Datentype: CUnit&
Legt die Einheit fest, in der die Ungenauigkeit auszugeben ist.

i_iResSubStrVisiblility (IN) Datentype: int (Default = SubStrVal | SubStrUnitSymbol)
Verodertes Bit-Feld über das festgelegt wird, welche Teilstrings die Angabe der Ungenauigkeit enthalten soll (siehe enum EPhysValSubStr).
Rückgabewert: ..... Datentype: QString

5.12.3.22. *toString(CUnit&, int, EUnitFind, int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Konvertiert den physikalischen Wert in einen String unter Berücksichtigung der Ungenauigkeit sowie der spezifizierten Format-Anweisungen.
Parameter:
i_unitVal (IN) Datentype: CUnit&
Legt die Einheit fest, in der der Wert auszugeben ist.
i_iValSubStrVisiblility (IN) Datentype: int
Verodertes Bit-Feld über das festgelegt wird, welche Teilstrings für den Größenwert auszugeben sind (siehe enum EPhysValSubStr).
i_unitFindRes (IN) Datentype: EUnitFind
Definiert, ob die Ungenauigkeit in der bestmöglichen Einheit auszugeben ist.
i_iResSubStrVisiblility (IN) Datentype: int (Default = SubStrVal | SubStrUnitSymbol)
Verodertes Bit-Feld über das festgelegt wird, welche Teilstrings die Angabe der Ungenauigkeit enthalten soll (siehe enum EPhysValSubStr).
Rückgabewert: ..... Datentype: QString

5.12.3.23. *toString(CUnit&, int, CUnit&, int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Konvertiert den physikalischen Wert in einen String unter Berücksichtigung der Ungenauigkeit sowie der spezifizierten Format-Anweisungen.
Parameter:
i_unitVal (IN) Datentype: CUnit&
Legt die Einheit fest, in der der Wert auszugeben ist.
i_iValSubStrVisiblility (IN) Datentype: int
Verodertes Bit-Feld über das festgelegt wird, welche Teilstrings für den Größenwert auszugeben sind (siehe enum EPhysValSubStr).
i_unitRes (IN) Datentype: CUnit&
Legt die Einheit fest, in der die Ungenauigkeit auszugeben ist.
i_iResSubStrVisiblility (IN) Datentype: int (Default = SubStrVal | SubStrUnitSymbol)
Verodertes Bit-Feld über das festgelegt wird, welche Teilstrings die Angabe der Ungenauigkeit enthalten soll (siehe enum EPhysValSubStr).
Rückgabewert: ..... Datentype: QString

5.12.3.24. *toString(const SValueFormatProvider&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:

Konvertiert den physikalischen Wert in einen String unter Berücksichtigung der Ungenauigkeit sowie der spezifizierten Format-Anweisungen.
Parameter:
i_valueFormat (IN) Datentype: SValueFormatProvider&
Die übergebene Struktur erlaubt sehr viele Freiräume bei der Festlegung der Formatierung (siehe struct SValueFormatProvider).
Rückgabewert: .....Datentype: QString

5.12.3.25. *convertValue(CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Konvertiert den physikalischen Wert in die gewünschte Einheit.
Parameter:
i_pUnitDst (IN) Datentype: CUnit*
Gewünschte Zieleinheit.

5.12.3.26. *convertValue(CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Konvertiert den physikalischen Wert in die gewünschte Einheit.
Parameter:
i_unitDst (IN) Datentype: CUnit&
Gewünschte Zieleinheit.

5.12.3.27. *operator == (const CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.28. *operator != (const CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.29. *operator < (const CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.30. *operator > (const CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.31. *operator <= (const CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.32. *operator >= (const CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht zwei Instanzen der Klasse. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.33. *operator = (const CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Zuweisungungsoperator.
Rückgabewert: .....Datentype: CPhysVal&

5.12.3.34. *operator + (const CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Addiert zwei Instanzen der Klasse. Die Werte sowie deren Ungenauigkeiten werden aufsummiert, die Einheiten ensprechend konvertiert.
Rückgabewert: .....Datentype: CPhysVal

5.12.3.35. *operator += (const CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Addiert zwei physikalische Werte und weist das Ergebnis der Instanz zu. Die Werte sowie deren Ungenauigkeiten werden aufsummiert, die Einheiten ensprechend konvertiert.
Rückgabewert: .....Datentype: CPhysVal&

5.12.3.36. *operator - (const CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Subtrahiert einen Größenwert. Der rechtsseitige Operator wird subtrahiert, deren Ungenauigkeiten werden aufsummiert, die Einheiten ensprechend konvertiert.
Rückgabewert: .....Datentype: CPhysVal

5.12.3.37. *operator -= (const CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Subtrahiert einen Größenwert und weist das Ergebnis der Instanz zu. Die Ungenauigkeiten werden aufsummiert, die Einheiten ensprechend konvertiert.
Rückgabewert: .....Datentype: CPhysVal&

5.12.3.38. *operator == (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht einen physikalischen Wert mit einem double Wert. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.39. *operator != (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht einen physikalischen Wert mit einem double Wert. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.40. *operator < (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:

Vergleicht einen physikalischen Wert mit einem double Wert. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.41. *operator > (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht einen physikalischen Wert mit einem double Wert. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.42. *operator <= (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht einen physikalischen Wert mit einem double Wert. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.43. *operator >= (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht einen physikalischen Wert mit einem double Wert. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.44. *operator = (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Weist dem Größenwert den double Wert zu.
Rückgabewert: .....Datentype: CPhysVal&

5.12.3.45. *operator + (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Addiert einen double Wert zu einer physikalischen Größe. Die Werte werden addiert, die Ungenauigkeit bleibt unverändert.
Rückgabewert: .....Datentype: CPhysVal

5.12.3.46. *operator += (double)*

Besitzerbereich: Instanz

Sichtbarkeit: öffentlich
Beschreibung:
Addiert einen double Wert zu einer physikalischen Größe und weist das Ergebnis der Instanz zu. Die Werte werden summiert, die Ungenauigkeit bleibt unverändert.
Rückgabewert: .....Datentype: CPhysVal&

5.12.3.47. *operator - (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Subtrahiert einen double Wert von einer physikalischen Größe. Der rechtsseitige Operator wird subtrahiert, die Ungenauigkeit bleibt unverändert.
Rückgabewert: .....Datentype: CPhysVal

5.12.3.48. *operator -= (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Subtrahiert einen double Wert von einer physikalischen Größe und weist das Ergebnis der Instanz zu. Die Ungenauigkeit bleibt unverändert.
Rückgabewert: .....Datentype: CPhysVal&

5.12.3.49. *operator * (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Multipliziert den physikalischen Wert mit einem double Wert. Die Ungenauigkeit wird ebenfalls multipliziert.
Rückgabewert: .....Datentype: CPhysVal&

5.12.3.50. *operator *= (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Multipliziert den physikalischen Wert mit einem double Wert und weist das Ergebnis der Intanz zu. Die Ungenauigkeit wird ebenfalls multipliziert.
Rückgabewert: .....Datentype: CPhysVal

5.12.3.51. *operator / (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Dividiert den physikalischen Wert durch einen double Wert. Die Ungenauigkeit wird multipliziert.
Rückgabewert: .....Datentype: CPhysVal&

5.12.3.52. *operator /= (double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Dividiert den physikalischen Wert durch einen double Wert und weist das Ergebnis der Intanz zu. Die Ungenauigkeit wird multipliziert.
Rückgabewert: .....Datentype: CPhysVal

5.12.3.53. *operator == (const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht eine physikalische Größe mit einem String. Dabei wird der String in einen physikalischen Größenwert konvertiert. Für den Inhalt des Strings gelten dieselben Bedingungen wie für den Konstruktor mit einem String als Eingabeparameter. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.54. *operator != (const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht eine physikalische Größe mit einem String. Dabei wird der String in einen physikalischen Größenwert konvertiert. Für den Inhalt des Strings gelten dieselben Bedingungen wie für den Konstruktor mit einem String als Eingabeparameter. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.55. *operator < (const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht eine physikalische Größe mit einem String. Dabei wird der String in einen physikalischen Größenwert konvertiert. Für den Inhalt des Strings gelten dieselben Bedingungen wie für den Konstruktor mit einem String als Eingabeparameter. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.56. *operator > (const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Vergleicht eine physikalische Größe mit einem String. Dabei wird der String in einen physikalischen Größenwert konvertiert. Für den Inhalt des Strings gelten dieselben Bedingungen wie für den Konstruktor mit einem String als Eingabeparameter. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.57. *operator <= (const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht eine physikalische Größe mit einem String. Dabei wird der String in einen physikalischen Größenwert konvertiert. Für den Inhalt des Strings gelten dieselben Bedingungen wie für den Konstruktor mit einem String als Eingabeparameter. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.58. *operator >= (const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Vergleicht eine physikalische Größe mit einem String. Dabei wird der String in einen physikalischen Größenwert konvertiert. Für den Inhalt des Strings gelten dieselben Bedingungen wie für den Konstruktor mit einem String als Eingabeparameter. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: bool

5.12.3.59. *operator = (const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Zuweisungsoperator. Dabei wird der String in einen physikalischen Größenwert konvertiert. Für den Inhalt des Strings gelten dieselben Bedingungen wie für den Konstruktor mit einem String als Eingabeparameter. Eine Gleichheit der Instanzen besteht dann, wenn ihre Werte (ggf. zuvor in die gleiche Einheit konvertiert) gleich sind.
Rückgabewert: .....Datentype: CPhysVal&

5.12.3.60. *operator + (const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Addiert einen String zu einer physikalischen Größe. Dabei wird der String in einen physikalischen Größenwert konvertiert. Für den Inhalt des Strings gelten dieselben

Bedingungen wie für den Konstruktor mit einem String als Eingabeparameter. Die Werte sowie deren Ungenauigkeiten werden aufsummiert, die Einheiten ensprechend konvertiert.
Rückgabewert: .....Datentype: CPhysVal

5.12.3.61. *operator += (const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Addiert einen String zu einer physikalischen Größe und weist das Ergebnis der Instanz zu. Für den Inhalt des Strings gelten dieselben Bedingungen wie für den Konstruktor mit einem String als Eingabeparameter. Die Werte sowie deren Ungenauigkeiten werden aufsummiert, die Einheiten ensprechend konvertiert.
Rückgabewert: .....Datentype: CPhysVal&

5.12.3.62. *operator - (const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Subtrahiert einen String von einer physikalischen Größe. Dabei wird der String in einen physikalischen Größenwert konvertiert. Für den Inhalt des Strings gelten dieselben Bedingungen wie für den Konstruktor mit einem String als Eingabeparameter. Die Ungenauigkeiten werden aufsummiert, die Einheiten ensprechend konvertiert.
Rückgabewert: .....Datentype: CPhysVal

5.12.3.63. *operator -= (const QString&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Subtrahiert einen String von einer physikalischen Größe und weist das Ergebnis der Instanz zu. Dabei wird der String in einen physikalischen Größenwert konvertiert. Für den Inhalt des Strings gelten dieselben Bedingungen wie für den Konstruktor mit einem String als Eingabeparameter. Die Ungenauigkeiten werden aufsummiert, die Einheiten ensprechend konvertiert.
Rückgabewert: .....Datentype: CPhysVal&

5.13. Klasse *CPhysValArr*

Häufig werden Array's von Messwerten oder Einstellparametern benötigt, in denen sich die Einheiten der einzelnen Werte sowie deren Ungenauigkeiten nicht voneinander unterscheiden. Für diesen Anwendungsfall wurde die Klasse *CPhysValArr* implementiert, die ein Array von double Werten verwaltet und eine Einheit sowie eine Ungenauigkeit, die für alle Werte gleich ist.

5.13.1. Attribute

m_pUnitGrp Datentype: CUnitGrp*
Verweis auf die Größenart der physikalischen Werte.
m_pUnit Datentype: CUnit*
Verweis auf die Einheit der physikalischen Werte.
m_validity Datentype: EValidity
Gibt an, ob die Werte gültig sind. Das Validity Flag wird in den Konstruktoren auf Invalid gesetzt. Es muss explizit gesetzt werden.
m_uValCount Datentype: unsigned int
Anzahl der Werte.
m_uValArrLen Datentype: unsigned int
Länge des Arrays, in dem die Werte abgelegt werden. Die Länge ist immer >= *ValCount*. Werden dem Array Werte hinzugefügt, wird die Größe des Arrays solange verdoppelt, bis eine maximale Anzahl neu anzulegender Elemente überschritten wird. Ist z.B. die maximale Anzahl neu anzulegende Elemente = 100 und beträgt die aktuelle Array-Länge = 200, wird das Array nur auf 300 Elemente vergrößert.
m_physValRes Datentype: CPhysValRes
Ungenaugkeit der physikalischen Werte.

5.13.2. Konstruktoren und Destruktor

5.13.2.1. *CPhysValArr(EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Werte jedoch noch nicht gültig sind.
Parameter:
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.13.2.2. *CPhysValArr(CUnitGrp&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Werte jedoch noch ungültig sind. Die Größenart wird bereits beim Anlegen der Instanz festgelegt.
Parameter:
i_unitGrp (IN) Datentype: CUnitGrp&
Referenz auf die Größenart der Werte.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.13.2.3. *CPhysValArr(CUnit*, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Erzeugt eine Instanz der Klasse, deren Werte jedoch noch ungültig sind. Die Einheit der Werte wird bereits beim Anlegen der Instanz festgelegt.
Parameter:
i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit der Werte (darf auch NULL sein).
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.13.2.4. *CPhysValArr(CUnit&, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Werte jedoch noch ungültig sind. Die Einheit der Werte wird bereits beim Anlegen der Instanz festgelegt.
Parameter:
i_unit (IN) Datentype: CUnit&
Referenz auf die Einheit der Werte.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.13.2.5. *CPhysValArr(const CPhysValArr&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Copy Konstruktor. Einheit, Validity und Ungenauigkeit wird übernommen. Die Werte werden kopiert und in ein Array mit denselben Dimensionen übernommen.
Parameter:
i_physValArr (IN) Datentype: const CPhysValArr&
Referenz auf das zu kopierende Array physikalischer Werte.

5.13.2.6. *CPhysValArr(CUnit*, unsigned int, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Werte jedoch noch ungültig sind. Die Einheit der Werte wird bereits beim Anlegen der Instanz festgelegt. Es wird ein Array mit der gewünschten Anzahl Elementen reserviert, das mit Nullen vorbelegt wird.
Parameter:
i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit der Werte (darf auch NULL sein).
i_uValCount (IN) Datentype: unsigned int
Gewünschte Größe des Arrays.

i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.13.2.7. *CPhysValArr(Cunit&, unsigned int, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Werte jedoch noch ungültig sind. Die Einheit der Werte wird bereits beim Anlegen der Instanz festgelegt. Es wird ein Array mit der gewünschten Anzahl Elementen reserviert, das mit Nullen vorbelegt wird.
Parameter:
i_unit (IN) Datentype: CUnit&
Referenz auf die Einheit der Werte.
i_uValCount (IN) Datentype: unsigned int
Gewünschte Größe des Arrays.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.13.2.8. *CPhysValArr(CUnit*, unsigned int, double*, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Werte jedoch noch ungültig sind. Die Einheit der Werte wird bereits beim Anlegen der Instanz festgelegt. Es wird ein Array mit der gewünschten Anzahl Elementen reserviert und die übergebenen Werte in dieses Array kopiert. Das Validity Flag muss explizit über *setValidity* gesetzt werden.
Parameter:
i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit der Werte (darf auch NULL sein).
i_uValCount (IN) Datentype: unsigned int
Gewünschte Größe des Arrays.
i_pfValues (IN) Datentype: double*
Verweis auf Array mit den zu übernehmenden Werten. Das Array muss mindestens ValCount Elemente besitzen.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.13.2.9. *CPhysValArr(Cunit&, unsigned int, double*, EResType)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Erzeugt eine Instanz der Klasse, deren Werte jedoch noch ungültig sind. Die Einheit der Werte wird bereits beim Anlegen der Instanz festgelegt. Es wird ein Array mit der

gewünschten Anzahl Elementen reserviert und die übergebenen Werte in dieses Array kopiert.
Parameter:
i_unit (IN) Datentype: CUnit&
Verweis auf die Einheit der Werte (darf auch NULL sein).
i_uValCount (IN) Datentype: unsigned int
Gewünschte Größe des Arrays.
i_pfValues (IN) Datentype: double*
Verweis auf Array mit den zu übernehmenden Werten. Das Array muss mindestens ValCount Elemente besitzen.
i_resType (IN) Datentype: EResType (Default = EResTypeResolution)
Legt fest, ob es sich bei der Genauigkeitsangabe um eine Auflösung oder eine Messunsicherheit handelt.

5.13.3. Operationen

5.13.3.1. *isValid*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Zeigt an, ob die Werte im Array gültig sind.
Rückgabewert: .....Datentype: bool

5.13.3.2. *setValidity(EValidity)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt das Validity Flag des Arrays (siehe enum EValidity).
Parameter:
i_validity (IN) Datentype: EValidity
Validity Flag des Arrays.

5.13.3.3. *getValidity*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt das Validity Flag des Arrays zurück (siehe enum EValidity).
Rückgabewert: .....Datentype: EValidity

5.13.3.4. *setUnitGroup(CUnitGrp*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Legt die Größenart der Werte fest.
Parameter:

i_pUnitGrp (IN) Datentype: CUnitGrp*
Verweis auf die Größenart (darf auch NULL sein)..

5.13.3.5. *getUnitGroup*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die Größenart der Werte zurück.
Rückgabewert: .....Datentype: CUnitGrp*

5.13.3.6. *setUnit(CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Legt die Einheit der Werte fest.
Parameter:
i_pUnitGrp (IN) Datentype: CUnitGrp*
Verweis auf die Einheit (darf auch NULL sein).

5.13.3.7. *getUnit*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die Einheit der Werte zurück.
Rückgabewert: .....Datentype: CUnit*

5.13.3.8. *getValCount*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die Anzahl der Werte im Array zurück. Die Werte können aber Invalid sein, wenn das Array angelegt, aber noch nicht mit Werten beschrieben wurden.
Rückgabewert: .....Datentype: unsigned int

5.13.3.9. *clear*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den für die Werte allokierten Speicher frei und setzt die Länge des Arrays auf Null.

5.13.3.10. *getPhysVal(unsigned int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:

Gibt den physikalischen Wert am gewünschten Index zurück. Einheit und Ungenauigkeit werden mit übernommen.
Parameter:
i_udx (IN) Datentype: unsigned int
Index innerhalb des Arrays, dessen Wert ausgelesen werden soll.
Rückgabewert: .....Datentype: CPhysVal

5.13.3.11. *getVal(unsigned int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Wert am gewünschten Index in der Einheit zurück, die im Array aktuell für alle Werte gilt.
Parameter:
i_udx (IN) Datentype: unsigned int
Index innerhalb des Arrays, dessen Wert ausgelesen werden soll.
Rückgabewert: .....Datentype: double

5.13.3.12. *getVal(unsigned int, CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Wert am gewünschten Index in der gewünschten Einheit zurück.
Parameter:
i_udx (IN) Datentype: unsigned int
Index innerhalb des Arrays, dessen Wert ausgelesen werden soll.
i_pUnit (IN) Datentype: CUnit*
Gewünschte Einheit, in der der Wert ausgelesen werden soll. Wird NULL übergeben, wird der Wert in der Einheit zurückgegeben, die aktuell im Array für alle Werte gilt.
Rückgabewert: .....Datentype: double

5.13.3.13. *getVal(unsigned int, CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt den Wert am gewünschten Index in der gewünschten Einheit zurück.
Parameter:
i_udx (IN) Datentype: unsigned int
Index innerhalb des Arrays, dessen Wert ausgelesen werden soll.
i_unit (IN) Datentype: CUnit&
Gewünschte Einheit, in der der Wert ausgelesen werden soll.
Rückgabewert: .....Datentype: double

5.13.3.14. *values(unsigned int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Gibt einen Verweis auf die Werte im Array beginnend ab dem gewünschten Start-Index zurück.
Parameter:
i_udxStart (IN) Datentype: unsigned int
Rückgabewert: Datentype: double*

5.13.3.15. *appendVal(double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Fügt einen Wert am Ende des Arrays an.
Parameter:
i_fVal (IN) Datentype: double

5.13.3.16. *appendVal(double, CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Fügt einen Wert am Ende des Arrays an. Ggf. wird der Wert in die aktuelle Einheit des Arrays konvertiert. Das Validity-Flag wird nicht gesetzt.
Parameter:
i_fVal (IN) Datentype: double
i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit des hinzuzufügenden Wertes. Wird NULL übergeben, wird der Wert nicht konvertiert.

5.13.3.17. *appendVal(double, CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Fügt einen Wert am Ende des Arrays an. Ggf. wird der Wert in die aktuelle Einheit des Arrays konvertiert. Das Validity-Flag wird nicht gesetzt.
Parameter:
i_fVal (IN) Datentype: double
i_unit (IN) Datentype: CUnit&
Referenc auf die Einheit des hinzuzufügenden Wertes. Ggf. wird der Wert in die Einheit des Arrays konvertiert.

5.13.3.18. *appendVal(CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Fügt einen physikalischen Wert am Ende des Arrays an. Ggf. wird der Wert in die aktuelle Einheit des Arrays konvertiert. Das Validity-Flag wird nicht gesetzt.
Parameter:

i_pyhsVal (IN) Datentype: CPhysVal

5.13.3.19. *removeVal(unsigned int)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Entfernt einen Wert am gewünschten Index aus dem Array. Die Werte nach dem entfernten Index werden eine Stelle „nach oben“ verschoben. Das Validity-Flag wird nicht gesetzt.
Parameter:
i_udx (IN) Datentype: unsigned int
Index innerhalb des Arrays, dessen Wert entfernt werden soll.

5.13.3.20. *setVal(unsigned int, double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt den Wert am gewünschten Index. Liegt der übergebene Index hinter der aktuellen Array-Länge, wird das entsprechend Array vergrößert und die Lücke mit Nullen initialisiert. Das Validity-Flag wird nicht gesetzt.
Parameter:
i_udx (IN) Datentype: unsigned int
Index, an dem der Wert gesetzt werden soll.
i_fVal (IN) Datentype: double

5.13.3.21. *setVal(unsigned int, double, CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt den Wert am gewünschten Index. Ggf. wird der Wert in die aktuelle Einheit des Arrays konvertiert. Liegt der übergebene Index hinter der aktuellen Array-Länge, wird das entsprechend Array vergrößert und die Lücke mit Nullen initialisiert. Das Validity-Flag wird nicht gesetzt.
Parameter:
i_udx (IN) Datentype: unsigned int
Index, an dem der Wert gesetzt werden soll.
i_fVal (IN) Datentype: double
i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit des hinzuzufügenden Wertes. Wird NULL übergeben, wird der Wert nicht konvertiert.

5.13.3.22. *setVal(unsigned int, double, CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt den Wert am gewünschten Index. Ggf. wird der Wert in die aktuelle Einheit des Arrays konvertiert. Liegt der übergebene Index hinter der aktuellen Array-Länge, wird das

entsprechend Array vergrößert und die Lücke mit Nullen initialisiert. Das Validity-Flag wird nicht gesetzt.

Parameter:

i_udx (IN) Datentype: unsigned int
Index, an dem der Wert gesetzt werden soll.

i_fVal (IN) Datentype: double

i_unit (IN) Datentype: CUnit&
Verweis auf die Einheit des hinzuzufügenden Wertes.

5.13.3.23. *setVal(unsigned int, CPhysVal&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Setzt den Wert am gewünschten Index. Ggf. wird der Wert in die aktuelle Einheit des Arrays konvertiert. Liegt der übergebene Index hinter der aktuellen Array-Länge, wird das entsprechend Array vergrößert und die Lücke mit Nullen initialisiert. Das Validity-Flag wird nicht gesetzt.

Parameter:

i_udx (IN) Datentype: unsigned int
Index, an dem der Wert gesetzt werden soll.

i_physVal (IN) Datentype: CPhysVal&

5.13.3.24. *setValues(unsigned int, unsigned int, double*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Übernimmt das übergebene Werte Array in das aktuelle Array. Überlappen sich der übergebene Index-Bereich mit dem aktuellen Index-Bereich im Array, werden die Werte im Array überschrieben. Ragt der übergebene Index Bereich über den aktuellen Index-Bereich des Arrays hinaus, wird das Array vergrößert. Eine dabei ggf. entstandene Lücke wird mit Nullen gefüllt. Das Validity-Flag wird nicht gesetzt.

Parameter:

i_udxStart (IN) Datentype: unsigned int
Start-Index, ab dem beginnend die übergebenen Werte in das Array kopiert werden sollen.

i_uValCount (IN) Datentype: unsigned int
Anzahl der Elemente, die übernommen werden sollen. Das übergebenen Array muss mindestens ValCount Elemente besitzen.

i_pfValues (IN) Datentype: double*
Verweis auf die zu übernehmenden Werte.

5.13.3.25. *setValues(unsigned int, unsigned int, double*, CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Übernimmt das übergebene Werte Array in das aktuelle Array. Ggf. werden die übergebenen Werte in die aktuelle Einheit des Arrays konvertiert. Die Konvertierung muss

Wert für Wert erfolgen und ist deshalb rechenintensiv. Überlappen sich der übergebene Index-Bereich mit dem aktuellen Index-Bereich im Array, werden die Werte im Array überschrieben. Ragt der übergebene Index Bereich über den aktuellen Index-Bereich des Arrays hinaus, wird das Array vergrößert. Eine dabei ggf. entstandene Lücke wird mit Nullen gefüllt. Das Validity-Flag wird nicht gesetzt.

Parameter:

i_udxStart (IN) Datentype: unsigned int
Start-Index, ab dem beginnend die übergebenen Werte in das Array kopiert werden sollen.

i_uValCount (IN) Datentype: unsigned int
Anzahl der Elemente, die übernommen werden sollen. Das übergebenen Array muss mindestens ValCount Elemente besitzen.

i_pfValues (IN) Datentype: double*
Verweis auf die zu übernehmenden Werte.

i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit der übergebenen Werte. Wird NULL übergeben, werden die Werte nicht konvertiert.

5.13.3.26. *setValues(unsigned int, unsigned int, double*, CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Übernimmt das übergebene Werte Array in das aktuelle Array. Ggf. werden die übergebenen Werte in die aktuelle Einheit des Arrays konvertiert. Die Konvertierung muss Wert für Wert erfolgen und ist deshalb rechenintensiv. Überlappen sich der übergebene Index-Bereich mit dem aktuellen Index-Bereich im Array, werden die Werte im Array überschrieben. Ragt der übergebene Index Bereich über den aktuellen Index-Bereich des Arrays hinaus, wird das Array vergrößert. Eine dabei ggf. entstandene Lücke wird mit Nullen gefüllt. Das Validity-Flag wird nicht gesetzt.

Parameter:

i_udxStart (IN) Datentype: unsigned int
Start-Index, ab dem beginnend die übergebenen Werte in das Array kopiert werden sollen.

i_uValCount (IN) Datentype: unsigned int
Anzahl der Elemente, die übernommen werden sollen. Das übergebenen Array muss mindestens ValCount Elemente besitzen.

i_pfValues (IN) Datentype: double*
Verweis auf die zu übernehmenden Werte.

i_unit (IN) Datentype: CUnit&
Referenz auf die Einheit der übergebenen Werte.

5.13.3.27. *hasRes*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:

Gibt an, ob die Werte im Array eine Ungenauigkeit besitzen.

Rückgabewert: .....Datentype: bool

5.13.3.28. *getRes*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Gibt die Ungenauigkeit der Werte im Array zurück.
Rückgabewert: .....Datentype: CPhysValRes

5.13.3.29. *setRes(double)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt die Ungenauigkeit der Werte im Array.
Parameter:
i_fVal (IN) Datentype: double
Ungenauigkeitsangabe in der aktuellen Einheit des Arrays. Ein Wert = 0.0 besagt, dass die Werte keine Ungenauigkeit besitzen.

5.13.3.30. *setRes(double, CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt Einheit und Wert für die Ungenauigkeit der Werte im Array.
Parameter:
i_fVal (IN) Datentype: double
Ungenauigkeitsangabe in der aktuellen Einheit des Arrays. Ein Wert = 0.0 besagt, dass die Werte keine Ungenauigkeit besitzen.
i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit der übergebenen Ungenauigkeitswerts. Kann auch NULL sein. Die Einheit wird in jedem Fall für die Ungenauigkeit übernommen. Sie muss also der Größenart der Werte im Array entsprechen.

5.13.3.31. *setRes(double, CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich
Beschreibung:
Setzt Einheit und Wert für die Ungenauigkeit der Werte im Array.
Parameter:
i_fVal (IN) Datentype: double
Ungenauigkeitsangabe in der aktuellen Einheit des Arrays. Ein Wert = 0.0 besagt, dass die Werte keine Ungenauigkeit besitzen.
i_pUnit (IN) Datentype: CUnit*
Referenz auf die Einheit der übergebenen Ungenauigkeitswerts. Die Einheit wird in jedem Fall für die Ungenauigkeit übernommen. Sie muss also der Größenart der Werte im Array entsprechen.

5.13.3.32. *setRes(const CPhysValRes&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Setzt Einheit und Wert für die Ungenauigkeit der Werte im Array.

Parameter:
i_physValRes (IN) Datentype: const CPhysValRes&
Ungenauigkeitsangabe für die Werte im Array.

5.13.3.33. *convertValues(CUnit*)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Konvertiert die Werte im Array in die gewünschte Einheit. Die Einheit muss der aktuellen Größenart der Werte im Array entsprechen.

Parameter:
i_pUnit (IN) Datentype: CUnit*
Verweis auf die Einheit, in die die Werte zu konvertieren sind. Die Einheit muss der aktuellen Größenart der Werte im Array entsprechen.

5.13.3.34. *convertValues(CUnit&)*

Besitzerbereich: Instanz
Sichtbarkeit: öffentlich

Beschreibung:
Konvertiert die Werte im Array in die gewünschte Einheit. Die Einheit muss der aktuellen Größenart der Werte im Array entsprechen.

Parameter:
i_unit (IN) Datentype: CUnit&
Referenz auf die Einheit, in die die Werte zu konvertieren sind. Die Einheit muss der aktuellen Größenart der Werte im Array entsprechen.