Unit Conversion Library User's Manual

Version 1.0

January 2024

Table of Contents

Table of Figures

Table of Tables

Table of Listings

1.0 Introduction

Anyone who has written code, especially code for an engineering or scientific application, has likely encountered the need to convert one or more units of measurement into another equivalent set of units in the same or a different system of measurement. Examples include converting feet to meters, pounds to kilograms, gigahertz to megahertz, and so on. The task of converting units is often accomplished in an *ad hoc* manner: using magic numbers e.g. 0.3048 to convert feet to meters or multiplying by 1000 to convert gigahertz to megahertz; introducing macros or constants into the code e.g. `#define FEET_TO_METERS 0.3048`; writing a small specialized library of functions to perform the required conversions e.g. `double feet_to_meters(double x)`. All of these approaches have disadvantages. Magic numbers leave the code littered with perplexing constants whose purpose is at best difficult to decipher. Macros and constants provide (in theory) some description of purpose but lack consistency in naming and organization. Unit conversion functions require maintaining and updating a library of specialized functions. In all cases precision and accuracy is left to the discretion of the developer making reuse problematic. For example, one statute mile is exactly (by definition) 1.609344 kilometers which is often approximated as 1.6 kilometers, depending on the application.

The unit conversion library described by this user's manual seeks to solve most (if not all) of these problems. It is comprehensive in that (almost) all known units, ancient, historical, and modern are included, precision is uniform across units, access is standardized, and no maintenance is required. The library is open source with a GNU General Public License. As such, it is both transparent and free to use (see §2, **License** for details). Four programming languages are supported: C++, C#, Java, and Python. Windows binaries (lib and DLL files) for C++ and C#, (class files) for Java, and (pyc files) for Python are included. Complier versions are listed in the **Appendix**. The C++ code has not been complied or tested using GCC, but should compile and run under Linux (hopefully) without modification.

Examples of usage in all four languages is given in §4, **Quick Start**, and readers wishing less detail may go there directly to start using the library. Those wishing more information can continue reading. The organizational details of the software are described in §6, **Software**, and information about unit conversion in general is found in §5, **Unit Conversion**.

Some basic terminology is useful in understanding both the organization of the software and the underlying units included and supported. *Metrication* is the process of converting local or traditional units of measurement into the metric system. This process began in France in the 1790s and has continued to the present day. Currently over 95% of the world has made the transition to metric system or recognizes the metric system as the standard by which all units are defined. For example, the United States still uses Imperial units e.g. feet, pounds, etc. and these units are defined in terms of metric units. One foot is defined as exactly 0.3048 meters, one pound as exactly 0.45359237 kilograms, and so forth.

A *unit of measurement* is an agreed upon standard for measuring a physical quantity. Examples include length, weight, area, volume, temperature, etc. The agreed upon standard such as a bar of fixed length,

or a cylinder of fixed weight can be local, regional, national, or international. Obviously the more widely accepted the standard is the greater its utility and acceptance. A *system of units* is one or more units of measurement used together either locally, regionally, nationally, or globally for the measurement of physical quantities. Examples include the Imperial system, the SI system, cgs system, etc. A *unit type* refers to the physical quantity the unit measures. For example, length, mass, area, voltage, etc. A unit of measurement is uniquely identified by its type, system, and name. For example, dry volume, Imperial, cup uniquely identifies the unit 'cup' used in the United States for measuring dry goods such a flour.

Canonical Units of measurement are length, mass (weight), area, and volume. These units are common to all systems of measurement ancient, historical, and modern. *Ancient Systems* of measurement are those in existence before the common era and include, Greek, Roman, and Egyptian (to name a few) units. *Historical Systems* are those used by nations and other entities before the adoption of the metric system. Ancient systems consist exclusively of canonical units and historical systems mainly, but not exclusively, of canonical units. *Modern Systems* are those based on SI units.

SI units form the basis for all measurement systems. Ancient and historical units are expressed in terms of SI units using archeological and written documentation describing these units of measurement so that they can be expressed in the metric system. Thus, for example, a Hebrew cubit is 0.555 meters. As such every unit of measurement has an SI *value* or *conversion factor.* Specifying the unit's SI value allows it to be converted to SI units and by extension to any other equivalent unit of measurement.

Unit Equivalency refers to the *dimensions* of the unit. Seven fundamental and two supplemental dimensions are defined as shown in Table 1.1 along with two additional unofficial dimensions used in computing and counting. The dimensions of all other units of measurement are combinations of one or more of these dimensions. Two units of measurement are said to be *equivalent* if they have the same dimensions. Unit conversion between two units of measurement is only possible if two units are equivalent.

Table 1.1: Unit Dimensions

Physical Quantity	Dimension	SI Unit	SI Abrrevation
Fundamental Units			
Amount	N	mole	Mol
Electric Current Intensity	I	ampere	A
Length	L	meter	m
Luminous Intensity	J	candela	cd
Mass	M	kilogram	kg
Temperature	Θ	kelvin	K
Time	T	second	s
Supplemental Units			
Plane Angle	α	radian	rad
Solid Angle	Ω	steradian	sr
Additional (Unofficial) Units			
Unit of Information	B	byte	b
Quantity	O	count	n

The following notation is used to express a unit's dimension. The unit's component dimensions (as given in Table 1.1) are separated by a period. If a component dimension appears more than once, an exponent is used. Component dimension appearing in the denominator are separated from those in the numerator by a forward slash. All dimensions to the right of the forward slash are in the denominator and all those to the left in the numerator. For example, voltage has dimensions $M.L^2/T^3.I$ i.e. $\frac{ML^2}{T^2I}$ that is kilograms-meter-squared per second-cubed per ampere (in SI units) which is defined as a *volt*. Any unit of measurement having these exact same dimensions is equivalent and may be converted to volts using its SI *value*.

The seven fundamental SI units in Table 1.1 are defined in terms of seven physical constants as given in Table 1.2. Prior to 2019, these constants were measured in terms of the meters, second, kelvin, and so on. After 2019 these constants were given defined exact values and the seven fundaments SI units defined in terms of these constants, as shown in Table 1.2.

Table 1.2: Physical Constants for Defining SI Fundamental Units

Defining Constant	Symbol	Exact Value	SI Unit
Hyperfine Transition Frequency of Cs.	Δv_{Cs}	9,192,631,770 Hz	**Second**. The duration of 9192631770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium-133 atom.
Speed of Light	c	299,792,458 m/s	**Meter**. The distance travelled by light in vacuum in 1/299792458 seconds.
Plank Constant	h	6.62607015×10^{-34} J·s	**Kilogram**. Determined by Planck constant *h* (6.62607015×10^{-34} $kg\cdot m^2/s$), given the definitions of the meter and the second.
Elementary Charge	e	$1.602176634\times10^{-19}$ C	**Ampere**. The flow of exactly $1/1.602176634\times10^{-19}$ times the elementary charge *e* per second.
Boltzmann Constant	k	1.380649×10^{-23} J/K	**Kelvin**. Determined by Boltzmann constant *k* (1.380649×10^{-23} $kg\cdot m^2/s^2.K$), given the definition of the kilogram, meter, and second.
Avogadro Constant	N_A	6.02214076×10^{23} 1/mol	**Mole**. The amount of substance of exactly 6.02214076×10^{23} elementary entities.
Luminous Efficacy of 540 THZ Radiation	K_{cd}	683 J.Ω/W	**Candela**. The luminous intensity, in a given direction, of a source that emits monochromatic radiation of frequency 5.4×10^{14} hertz and that has a radiant intensity in that direction of 1/683 watt per steradian.

Since area and volume have dimensions L^2 and L^3, respectively, the canonical units are completely defined in terms of the meter and kilogram. The SI derived units, as given in Table 1.3 are used to measure electromagnetic fields, force, power, energy, radiation, and frequency. These quantities were observed to some extent by ancient cultures but remained essentially unknown or at least unmeasured or quantified until about the 15th century. These units are part of the SI system of measurement.

Table1.3: Derived Units

Unit	Abbreviation	Physical Measurement	Dimension
becquerel	Bq	Radioactivity	$1/T$
coulomb	C	Electric Charge	$I.T$
farad	F	Capacitance	$T^4.I^2/M.L^2$
gray	Gy	Absorbed Dose of Radiation	L^2/T^2
henry	H	Inductance	$M.L^2/T^2.I^2$
hertz	Hz	Frequency	$1/T$
joule	J	Energy	$M.L^2/T^2$
lumen	lm	Luminous Flux	$J.\Omega$
lux	Lx	Illuminance	$J.\Omega/L^2$
newton	N	Force	$M.L/T^2$
ohm	Ω	Resistance	$M.L^2/T^3.I^2$
pascal	Pa	Pressure	$M/L.T^2$
poiseuille	Po	Viscosity	$M/L.T$
siemens	S	Conductance	$T^3.I^2/M.L^2$
sievert	Sv	Dose Equivalent	L^2/T^2
tesla	T	Magnetic Flux Density	$M/T^2.I$
volt	V	Electric Potential	$M.L^2/T^3.I$
watt	W	Power	$M.L^2/T^3$
weber	Wb	Magnetic Flux	$M.L^2/T^2.I$

The SI system also includes prefixes that denote powers of ten of its fundamental and derived units. These are given in Table 1.4. Examples include centimeters (10^{-2} meters), megahertz (10^6 hertz), kilowatts (10^3 watts), and millivolts (10^{-3} volts). These prefixes can be applied to all the SI units.

Table 1.4: SI Prefixes

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

Conversion between equivalent units is always allowed even if the practical utility of doing such a conversion is dubious. For example, volume is divided into liquid and dry units of measurement. Converting gallons to bushels is allowed since both have dimensions of L^3. The utility of such a conversion however may have little practical value. Another example is converting hertz to becquerels. The former measures frequency and the latter radiation, but both have dimension of T^{-1}. Conversion between nonequivalent units (e.g. feet to kilograms) is of course, not allowed and the software checks the unit dimensions before attempting a conversion and generates an error if the units are not equivalent.

Cardarelli [1, 2] is perhaps the definitive and most comprehensive single source for units. His works claims over "10,000 precise conversion factors, and around 2,000 definitions of the units themselves". Other sources include [3-5], the internet, and Wikipedia. Cardarelli work is not without criticism (see [6], for example) but it remains widely cited and is a comprehensive single source for a vast number of units. Perhaps its biggest limitation is that unit values are given to only 10 decimal places which does not take advantage of the 15 decimal places of a double precision number on modern computers. The units in the unit conversion library come from a variety of sources and are given to 15 decimal places to provide maximum precision. For further discussion see §5, **Unit Conversion**.

2.0 License

The unit conversion software licensed under the GNU General Public License. A copy of this license is provided in a separate file and can also be found at https://www.gnu.org. In the case of the unit conversion software, anyone is free to use, incorporate, or distribute the software free of charge as described in the GNU General Public License. The software is open source and is provided without warranty of any kind. More specifically:

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

See the GNU General Public License for more details on warranty and for the entire licensing agreement.

PLEASE NOTE: Nothing in this user's manual shall be deemed to constitute or be interpreted to supersede or in any way invalidate or make inapplicable any provision in the GNU Public License provisions. In the event that any statement in the user's manual conflicts with the GNU Public License provisions, the GNU Public License provisions shall control at all times.

3.0 Organization

The Unit Conversion Library contains several components which taken together constitute the library (see Figure 3.1). The first is the Excel spreadsheet (workbook), `UnitTableX_01.xlsm`, containing all the units in the library. The second is the folder *UnitConversionLibrary* which contains the source code and binaries for incorporation into a project requiring unit conversion. The `Examples` folder contains examples in the four supported languages and is discussed in detail in §4, **Quick Start**. The examples can be compiled and run, if desired. Next is the `Test` folder which has both system and units tests for the code and which can also be run. The `GNU_License` folder contains the software license and the `UserManual` folder this document. The `Dev` folder contains Visual Studio solutions for compiling the library.

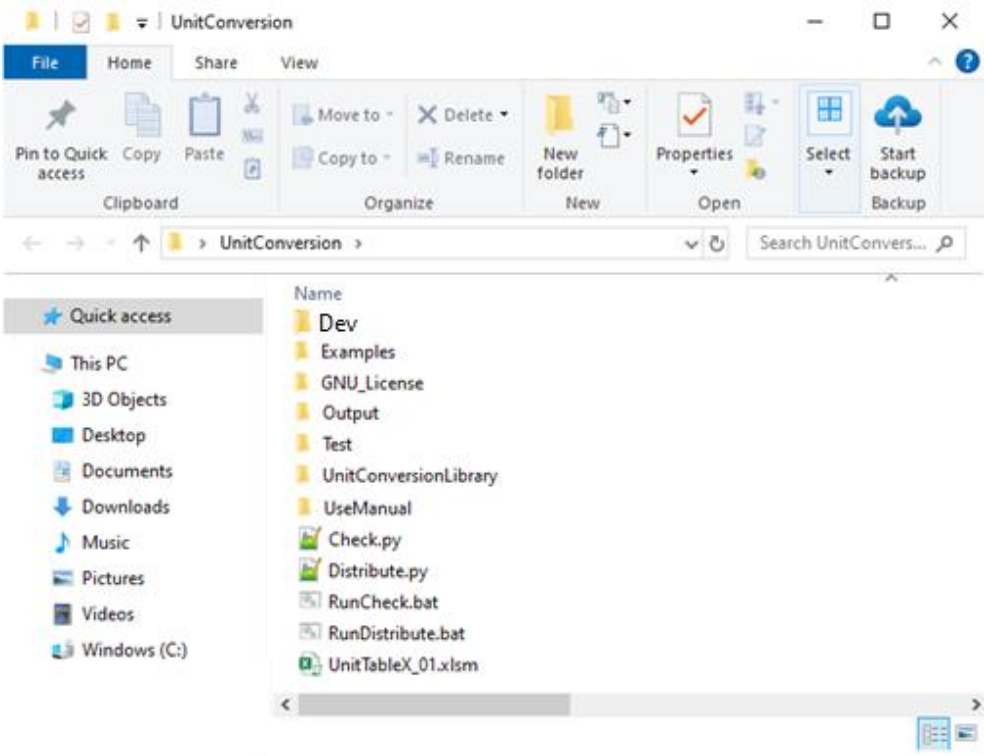

Figure 3.1: *Unit Conversion Library.* The library contains six folders at the top level and five files. The folder *UnitConversionLibary* contains the binaries and software. The file `UnitTableX_01.xlsm` is a spreadsheet containing the units contained in the library. The `Examples` folder contains usage examples in the four supported languages and the `Test` folder system and unit tests for the software. The `GNU_License` folder contains the software license.

The `Output` folder is empty and intended to hold the generated code from the spreadsheet (see §6, **Software**). The Python script `Check.py` is used to check the test output for errors (again, see §6). The second script, `Distribute.py`, is used to place the generated code in the `Output` folder into the library. The `RunCheck.bat` and `RunDistribute.bat` files run the check and distribute scripts, respectively on Windows.

4.0 Quick Start

This section is indented for those who wish to use the unit conversion library without reading through a "long and tedious" user's manual. It provides complete examples in the four supported programming languages listed in the **Introduction**. After reading this (intentionally) brief section it should be possible to use the library for many unit conversion tasks, albeit with a limited understanding of its full set of features. This section is divided into two parts: *Environment Setup* and *Software Usage*. For those familiar with setting up Visual Studio, Java class paths, and Python modules, the *Environment Setup* section may be skipped.

Environment Setup

For all four languages, the first step in setting up the environment is to place the folder *UnitConversionLibrary* within the project where it will be used. The Unit Conversion Library contains four folders, one for each language. Inside these folders are folders named *UnitConversion* and *Generated* which contains the library source code (and binaries for Java and Python). For C++ and C# there is an additional folder named *Libraries* which contains the binaries for those languages. A copy of the *UnitConversionLibrary* folder should be collocated with the project files where the library will be used.

For C++ users using Visual Studio, the include and link paths must be set. To set the include path, go to *Properties->C/C++->General* under *Additional Include Directories* and add the folder *UnitConversionLibrary / UnitConversion /Includes* (see Figure 4.1). Next the binary library and its path must be added to the linker portion of the project. Go to *Properties->Linker->Input* and add the file *UnitConversionLib.lib* to *Additional Dependencies* (see Figure 4.2). Finally go to *Properties->Linker-General* and add the path to the location of the folder containing the *UnitConversionLib.lib* file to *Additional Library Directories* (see Figure 4.3). There are two binaries, one compiled debug and the other release. They are located in *UnitConversionLibrary /Libraries/Debug* and *UnitConversionLibrary /Libraries/Release*. Choose the appropriate path for the project. Users of other systems, such as Linux, will need to compile the source files to generate the appropriate binaries using make-files or other equivalent mechanisms.

For C# users go *to Dependencies->Browse* and use the *Browse* button to add the file *UnitConversionLib.dll* to the project dependencies. See Figure 4.4. For both C# and C++, the source files are included in the *UnitConversionLibrary* directory but these files need not be included in the project (unless recompiling them is desired).

For Java users the path to the *UnitConversionLibrary* directory needs to be added to the CLASSPATH environment variable. For example:

```
set CLASSPATH=./;./UnitConversionLibrary;./UnitConversionLibrary/UnitConversion
```

works when the *UnitConversionLibrary* directory is located at the top level of the project directory. For other locations, the CLASSPATH will need to be adjusted accordingly to reflect the actual placement of *UnitConversionLibrary* directory in the project hierarchy.

The situation with python is similar. Set the `PYTHONPATH` environment variable, for example, to:

```
SET PYTHONPATH=./;./UnitConversionLibrary/
```

This works when the *UnitConversionLibrary* directory is located at the top level of the project directory. Again, this will need adjustment to reflect the actual placement of *UnitConversionLibrary* directory in the project hierarchy.

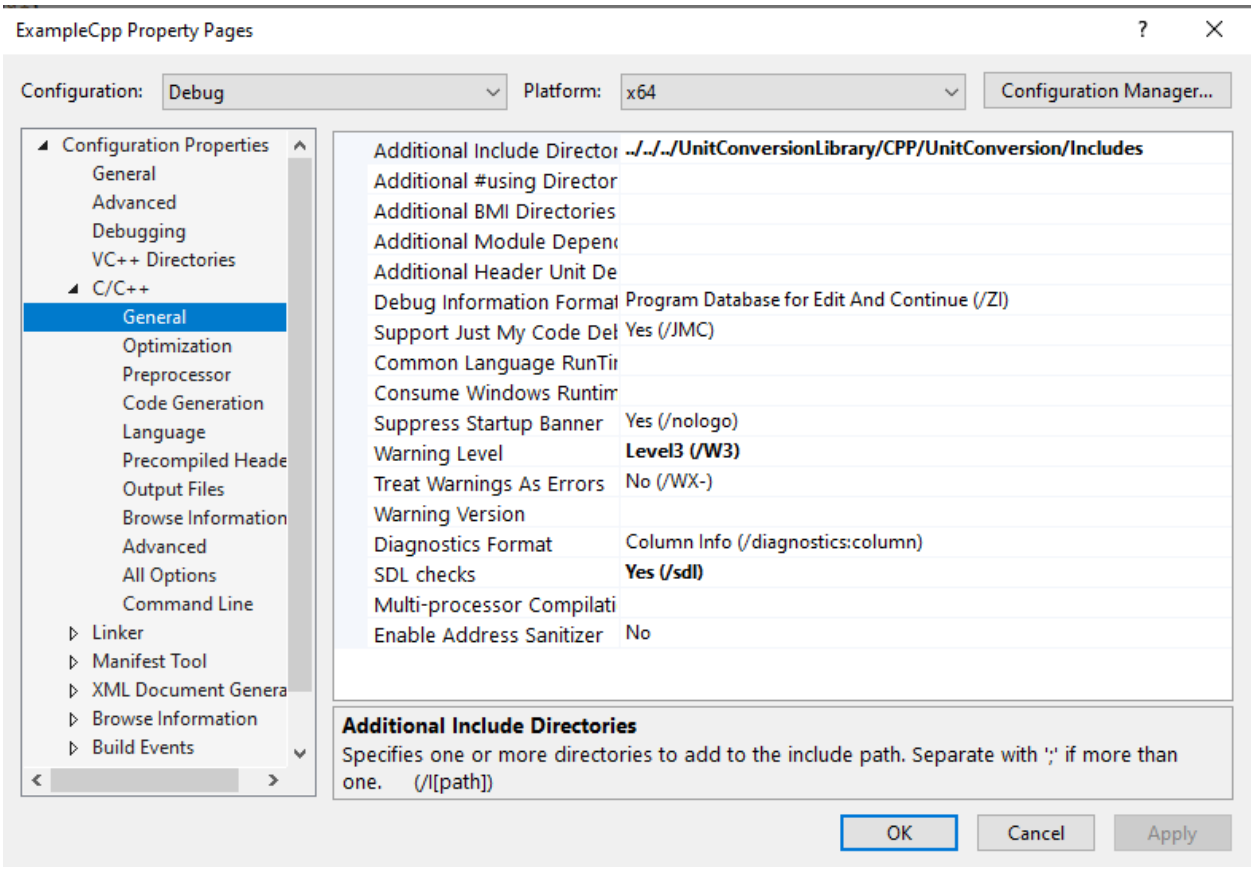

Figure 4.1: *Adding Path to Source Files.* The folder `UnitConversionLibrary` contains a folder with the header files required to access the Unit Conversion Library. The relative path will vary depending upon the chosen location for the folder `UnitConversionLibrary` with the project.

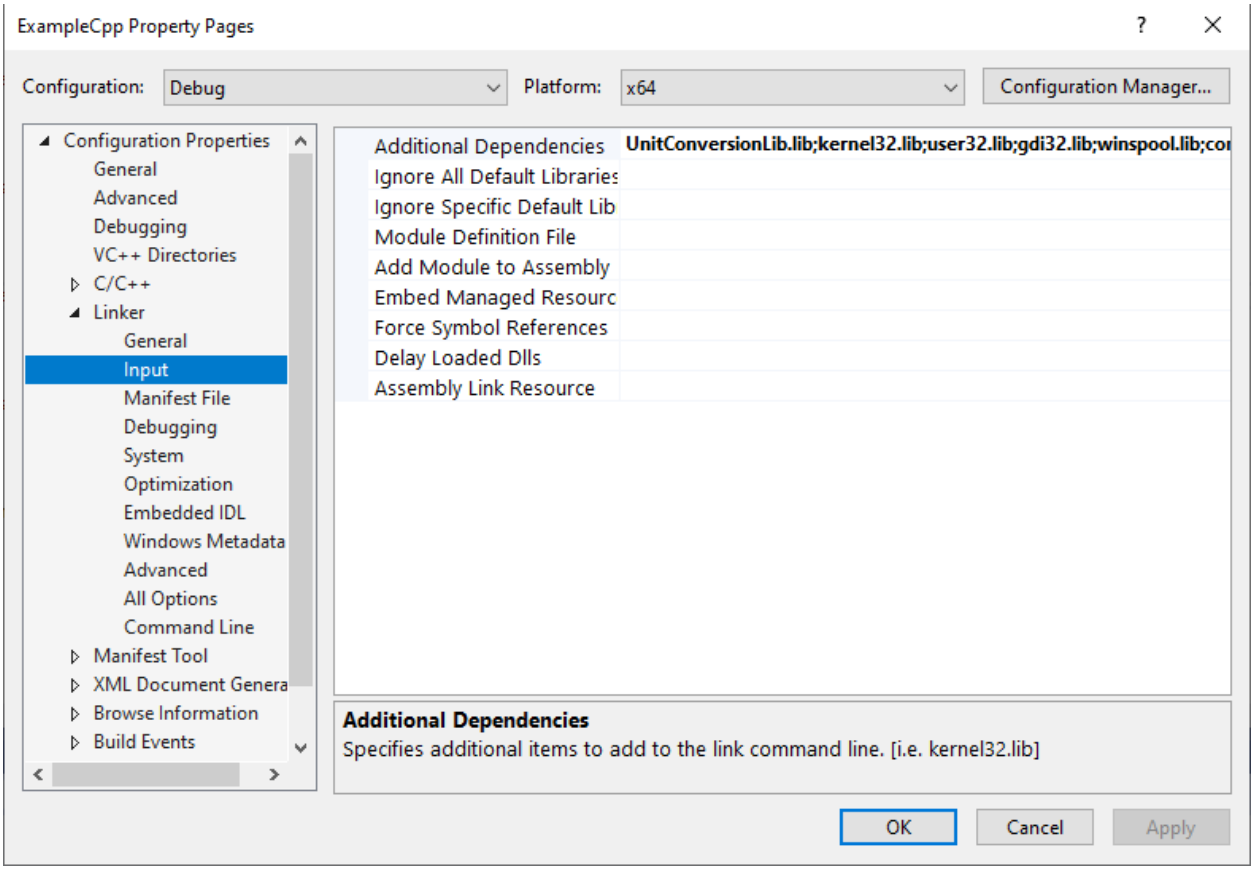

Figure 4.2: *Adding Library to Linker.* The file `UnitConversionLib.lib` is the binary library containing the complied Unit Conversion software. Add it to the list of inputs for the project.

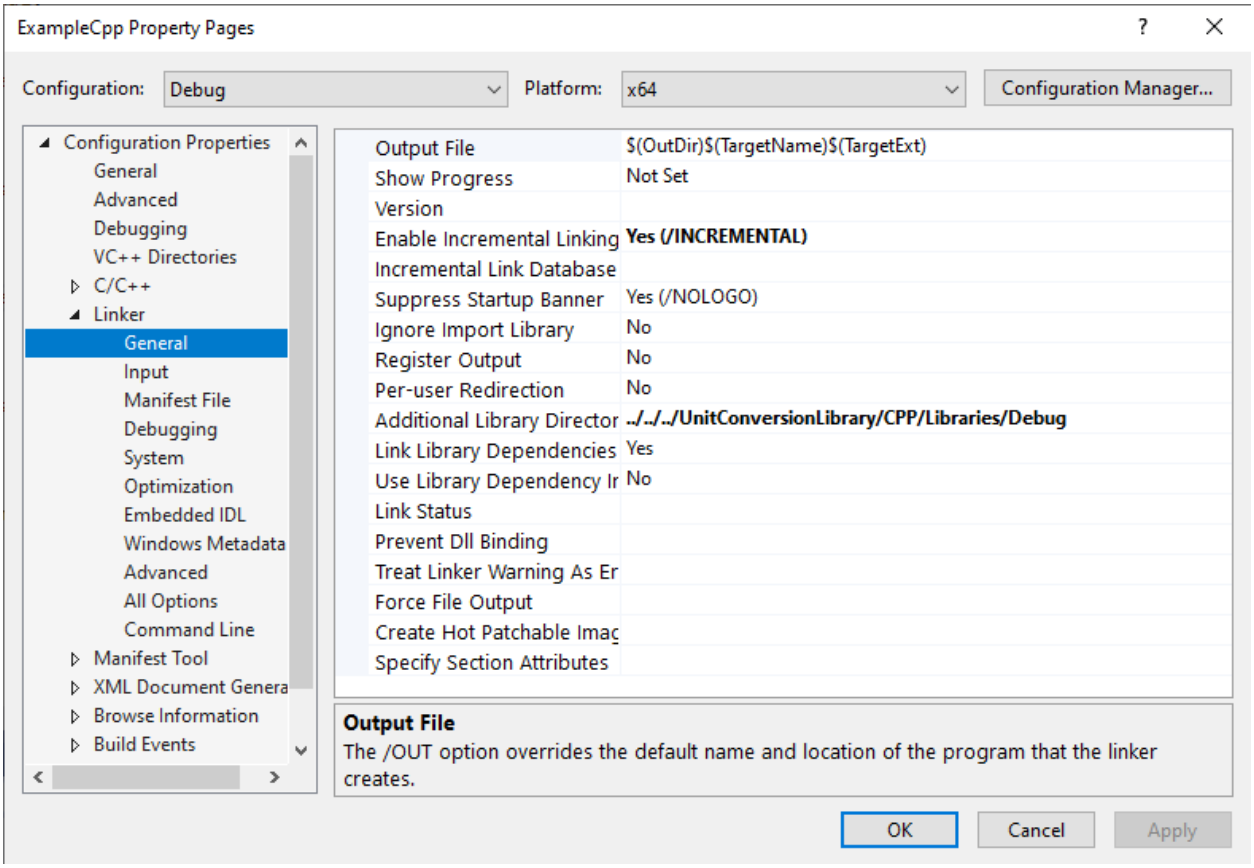

Figure 4.3: *Adding Path to Binaries.* There are two binaries, one compiled debug and the other release. They are located in *UnitConversionLib/CPP/Libraries/Debug* and *UnitConversionLibrary/CPP/Libraries/Release*. Choose the appropriate path for the project and add the path to Additional Library Directories.

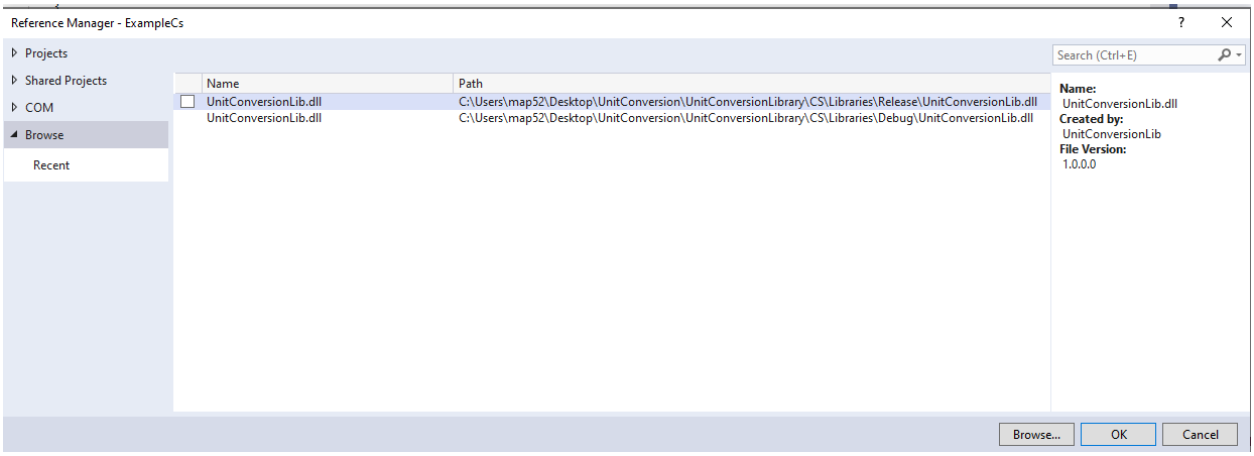

Figure 4.4: *5Adding C# Dependency.* There are two binaries, one compiled debug and the other release. They are located in *UnitConversionLibrary/CS/Libraries/Debug* and *UnitConversionLibrary/CS/Libraries/Release*. Choose the appropriate binary for the project and add the file `UnitConversionLib.dll` to the list of project dependencies.

Software Usage

Once the Unit Conversion Library has been integrated into the project (see above), it is ready to use. The class `UnitConversions` is a singleton the provides access to all unit conversions in the library. For all four languages the first step to call the (static) `Instance` method of this class to obtain the single instance to this class. Once an instance is obtained, it can be used to fetch a `Converter` object for each supported unit type. A list of unit types included in the library can be found in the Appendix. The converter object is used to convert equivalent units from one system to another or other equivalent units in the same system. Four example programs (one in each language) are provided. The listings and outputs of these programs are used in the following examples to illustrate usage.

C++ Example

Listing 4.1 shows an example of using the Unit Conversion Library in a simple C++ application. Referring to the listing, line 9 includes the header file `UnitConverions.hpp` which contains the interface to the library. Line 15 obtains a reference to the single instance of the library. This provides access to all the units in the library.

Lines 17-20 use the library instance from line 5 to obtain references to four converters contained in the library. Thes are length, mass, liquid volume, and temperature, respectively. Each converter allows unit conversions for its type. A complete list of types is given in the **Appendix**. Lines 24 and 25 use the length converter to convert 3.5 kilometers into meters and miles. Each converter has a default from-system and to-system. The from-system is the unit system of the unit being converted. The to-system is the system of the unit after conversion. A unit is uniquely identified by its type e.g. length, area, time, etc., its system e.g. SI, US, Imperial, UK, etc., and its name e.g. meters, feet, pound, kilogram, etc. In the case of length this is the SI system. Since kilometers and meters are both units in the SI system there is no need to specify the from-system and to-system when calling the `convert` method (line 24). However, since kilometers is a unit in the SI system and miles is a unit in the Imperial system, the from-system and to-system are specified when calling the `convert` method on line 25.

The converters have `fromSystem` and `toSystem` methods that can be called to obtain the current from-system and to-system names, respectively. These methods can also be used to change the from-system and to-system names. For more details see §6, **Software**.

Lines 33 and 34 of the listing repeat this exercise using the mass converter. In this case, pounds are converted to kilograms and Egyptian kantars. The call to the `convert` method is the same as described above. The first argument to the `convert` method is always the unit value to be converted. The second argument is the unit's name (in its current system). Next comes the name of the unit's (current) system. This is followed by the name of the unit in the desired (converted) system. Last comes the name of the system the unit is being converted into. The `convert` method returns the value of the unit in the new system and units. Double precision arithmetic is used in all conversions and the conversion factors (values) are stored in the library to 15 digits (base-10) for maximum precision.

Lines 42 and 43 convert liters to liquid gallons in both the US and UK system of units. Note that UK gallons are larger than US when viewing the output of the program. Finally, lines 51 and 52 convert 37° Celsius (body temperature) to Fahrenheit and Rankine. The output of the program is shown in Figure 4.5.

```cpp
////////////////////////////////////////////////////////////////////////////
//
// File: Example.cpp
//
// Program to demonstrate library usage.
//
////////////////////////////////////////////////////////////////////////////
#include "UnitConversions.hpp"
#include <iostream>

int main(int argc, char *argv[])
{
   std::cout << "Start Example" << std::endl << std::endl;

   UnitConversions& unitConversions = UnitConversions::Instance();

   Converter& length = unitConversions.converter("Length");
   Converter& mass   = unitConversions.converter("Mass");
   Converter& liquid = unitConversions.converter("Liquid");
   Converter& temp   = unitConversions.converter("Temperature");

   // Length conversion: convert kilometers to meters and miles.
   double km = 3.5;
   double m  = length.convert(km, "kilometer", "meter");
   double mi = length.convert(km, "kilometer", "SI", "mile", "Imperial");
   std::cout << "Length" << std::endl;
   std::cout << km << " kilometers = " << m  << " meters" << std::endl;
   std::cout << km << " kilometers = " << mi << " miles"  << std::endl;
   std::cout << std::endl;

   // Mass conversion. Convert pounds to kilograms and Egyptian kantar.
   double pounds = 100.0;
   double kgm    = mass.convert(pounds, "pound", "US", "kilogram", "SI");
   double kantar = mass.convert(pounds, "pound", "US", "kantar",   "Egyptian");
   std::cout << "Mass" << std::endl;
   std::cout << pounds << " pounds = " << kgm    << " kilograms" << std::endl;
   std::cout << pounds << " pounds = " << kantar << " kantar"    << std::endl;
   std::cout << std::endl;

   // Liquid volume conversion: convert liters to gallons
   double liters = 12.0;
   double gUS = liquid.convert(liters, "liter", "SI", "gallon", "US");
   double gUK = liquid.convert(liters, "liter", "SI", "gallon", "UK");
   std::cout << "Liquid Volume" << std::endl;
   std::cout << liters << " liters = " << gUS << " gallons (US)" << std::endl;
   std::cout << liters << " liters = " << gUK << " gallons (UK)" << std::endl;
   std::cout << std::endl;

   // Temperature Conversion: Celsius to Fahrenheit and Rankine
   double celsius    = 37.0;
   double fahrenheit = temp.convert(celsius, "Celsius", "INT", "Fahrenheit", "Imperial");
   double rankine    = temp.convert(celsius, "Celsius", "INT", "Rankine", "Imperial");
   std::cout << "Temperature" << std::endl;
   std::cout << celsius << " Celsius = " << fahrenheit << " Fahrenheit" << std::endl;
   std::cout << celsius << " Celsius = " << rankine    << " Rankine"    << std::endl;
   std::cout << std::endl;

   std::cout << "End Example" << std::endl;
}
// EOF
```

Listing 4.1: *C++ Example.* C++ example showing unit conversion of length, mass, liquid volume, and temperature.

C# Example

Listing 4.2 shows an example of using the Unit Conversion Library in a simple C# console application. Referring to the listing, line 9 has the using statement for the Unit Conversion Library. Line 19 obtains a reference to the single instance of the library. This provides access to all the units in the library. Lines 20-23 use the conversion library singleton to obtain references to four `Converters`. A converter is obtained for each unit type to be converted. In the case of the listing these are length, mass, liquid volume, and temperature.

Each `Converter` has a `convert` method that performs the unit conversion. Since a unit is uniquely identified by its type, system, and name, it is necessary to specify the unit's name and system both before and after the conversion. The unit's current system is termed the from-system and its system after conversion is termed the to-system The from-system and to-system may be the same or different. Of course, conversion is allowed only between units of the same type.

Each `Converter` has a default from-system and to-system. The names of these systems can be determined by calling the `fromSystem` and `toSystem` methods. These methods may also be used to change the from-system and to-system. Line 27 of the listing converts kilometers to meters. Both these units belong to the SI system which is the default from-system and to-system for length. As such, the names of the from-system and to-system may be omitted when using the `convert` method. On line 28 kilometers is converted to miles. Miles belongs to the Imperial system of units and as such, it is necessary to specify the names of both the from-system and to-system when calling the `convert` method. Alternatively, the statement:

```
length.toSystem("Imperial");
```

could have been inserted between lines 27 and 28 in which case, the from-system and to-system names could have been omitted from the call to the `convert` method.

Lines 35 and 36 converts pounds to kilograms and Egyptian kantars. An Egyptian kantar is the official Egyptian weight unit for measuring cotton and corresponds to the US hundredweight. The Unit Conversion Library supports conversion for units from about 80 countries. See the Appendix for a complete list of unit systems contained in the library. Lines 43 and 44 convert liters to US gallons and UK gallons. Note: a US gallon is about 80% of a UK gallon.

Lines 51 and 52 convert 37° Celsius (body temperature) to Fahrenheit and Rankine. The Rankine temperature scale, was established in 1859 by William John Macquorn Rankine (1820–1872). Like the Kelvin temperature scale, its zero is the theoretical temperature at which the molecules of matter have the absolute lowest energy. Unlike the Kelvin scale, the Rankine scale has the same unit size as the Fahrenheit. The two scales are related by the equation °R = °F + 459.67.

Figure 4.6 shows the output of the C# listing. Note: some unit conversions have been omitted from the listing for brevity but appear in the example program. A comparison of Figures 4.5 and 4.6 show identical outputs except for the precision of the printed results. The default precision of each language output was used.

```
01 ////////////////////////////////////////////////////////////////////////////
02 //
03 // File: Program.cs
04 //
04 // Program to demonstrate library usage.
06 //
07 ////////////////////////////////////////////////////////////////////////////
08 using System;
09 using UnitConversion;
10
11 namespace ExampleCs
12 {
13    class Program
14    {
15        static void Main(string[] args)
16        {
17            Console.WriteLine("Start Example");
18
19            UnitConversions unitConversions = UnitConversions.Instance();
20            Converter  length = unitConversions.converter("Length");
21            Converter  mass   = unitConversions.converter("Mass");
22            Converter  liquid = unitConversions.converter("Liquid");
23            Converter temp    = unitConversions.converter("Temperature");
24
25            // Length conversion: convert kilometers to meters and miles.
26            double km = 3.5;
27            double m  = length.convert(km, "kilometer", "meter");
28            double mi = length.convert(km, "kilometer", "SI", "mile", "Imperial");
29            Console.WriteLine("Length");
30            Console.WriteLine(km + " kilometers = " + m  + " meters" );
31            Console.WriteLine(km + " kilometers = " + mi + " miles");
32
33            // Mass conversion. Convert pounds to kilograms and Egyptian kantar.
34            double pounds = 100.0;
35            double kgm    = mass.convert(pounds, "pound", "US", "kilogram", "SI");
36            double kantar = mass.convert(pounds, "pound", "US", "kantar",   "Egyptian");
37            Console.WriteLine("Mass");
38            Console.WriteLine(pounds + " pounds = " + kgm    + " kilograms");
39            Console.WriteLine(pounds + " pounds = " + kantar + " kantar");
40
41            // Liquid volume conversion: convert liters to gallons
42            double liters = 12.0;
43            double gUS    = liquid.convert(liters, "liter", "SI", "gallon", "US");
44            double gUK    = liquid.convert(liters, "liter", "SI", "gallon", "UK");
45            Console.WriteLine("Liquid Volume");
46            Console.WriteLine(liters + " liters = " + gUS + " gallons (US)");
47            Console.WriteLine(liters + " liters = " + gUK + " gallons (UK)");
48
49            // Temperature Conversion: Celsius to Fahrenheit and Rankine
50            double cels       = 37.0;
51            double fahrenheit = temp.convert(cels, "Celsius", "INT", "Fahrenheit", "Imperial");
52            double rankine    = temp.convert(cels, "Celsius", "INT", "Rankine",    "Imperial");
53            Console.WriteLine("Temperature");
54            Console.WriteLine(cels + " Celsius = " + fahrenheit + " Fahrenheit");
55            Console.WriteLine(cels + " Celsius = " + rankine    + " Rankine");
56
57            Console.WriteLine("End Example");
58        }
59    }
60 }
61 // EOF
```

Listing 4.2: *C# Example.* C# example showing unit conversion of length, mass, liquid volume, and temperature.

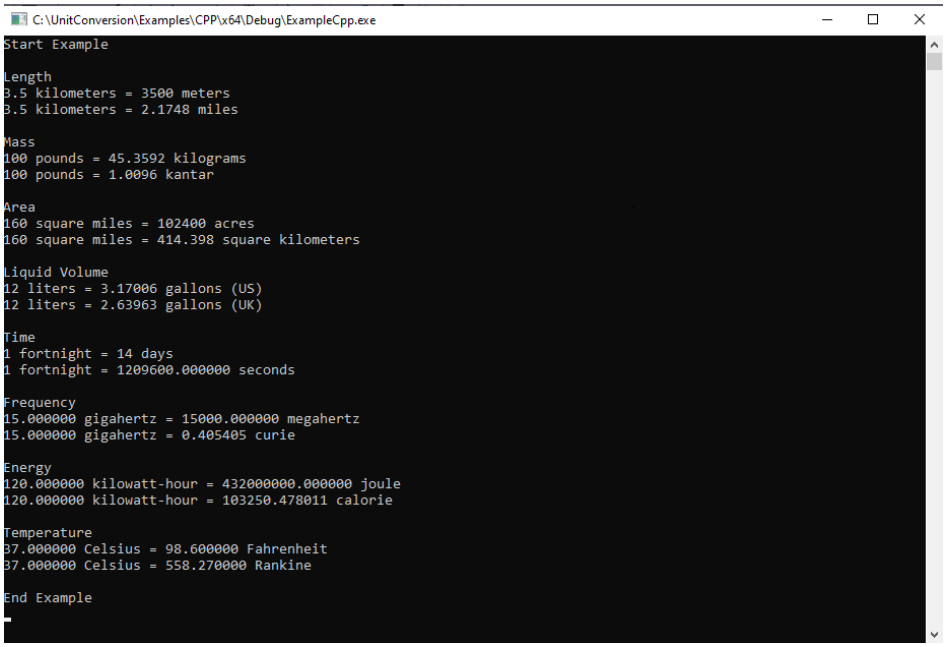

Figure 4.5: *C++ Example Output.* Three and a half kilometers is converted to meters and miles, one hundred pounds to kilograms and Egyptian kantars, one hundred and sixty square miles to acres and square kilometers, twelve liters to gallons, a fortnight to days and seconds, fifteen gigahertz to megahertz and curies, one hundred twenty kilowatt-hours to joules and calories, and thirty-seven degrees Celsius to Fahrenheit and Rankine.

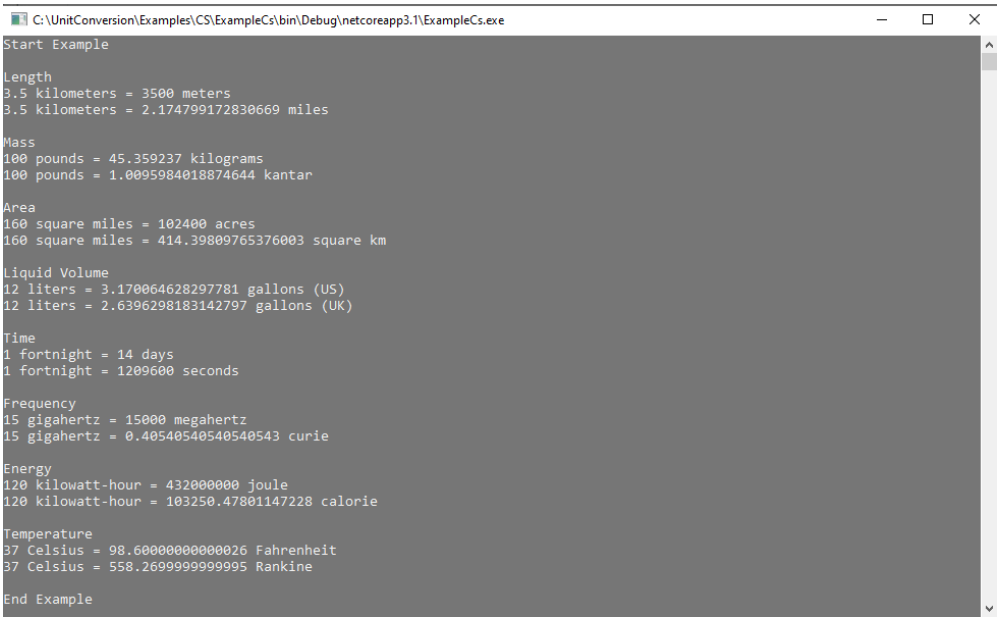

Figure 4.6: *C# Example Output.* Three and a half kilometers is converted to meters and miles, one hundred pounds to kilograms and Egyptian kantars, one hundred and sixty square miles to acres and square kilometers, twelve liters to gallons, a fortnight to days and seconds, fifteen gigahertz to megahertz and curies, one hundred twenty kilowatt-hours to joules and calories, and thirty-seven degrees Celsius to Fahrenheit and Rankine.

Java Example

Listing 4.3 shows an example of using the Unit Conversion Library in a simple Java application. In the example program, the Unit Conversion Library is located at the top level of the project directory. Lines 8 and 9 of the listing import the Unit Conversion Library and the converter for performing the unit conversions. Compiling the example program requires setting the CLASSPATH environment variable to include the path to the Unit Conversion Library, as described above.

Line 17 of the listing obtains a reference to the library. This refence is used to access all the supported unit conversions. Lines 18-21 use this reference to extract the desired `Converters` used to perform unit conversion. There is a unit converter in the library for each unit type. As noted in the **Introduction**, conversions are allowed only between units of the same type. In the case of the listing, four converters are used: one each for length, mass, liquid volume, and temperature.

On lines 25 and 26 the length converter is used to convert kilometers to meters and miles. Units are uniquely identified by their type, system, and name. A converter's `convert` method requires the unit name and system to perform the desired conversion. The name of the unit's system is termed the from-system and then name of the system of the converted unit is termed the to-system. The from-system and to-system may be the same or different. Each converter has a default from-system and to-system. The names of these systems may be obtained by calling the converter's `fromSystem` and `toSystem` methods. These methods can also be used to change the converter's from-system and to-system names.

In line 25, kilometers and meters are both units in the SI system, which is the default from-system and to-system for length conversions. As such, the names of the from-system and to-system may be omitted when calling the `convert` method. This is not the case for kilometers and miles on Line 26. Miles is a unit in the Imperial system and therefore the names of the from-system and to-system must be included in when calling the `convert` method. An alternative would be to set the to-system to be the Imperial system by inserting the following between lines 25 and 26:

```
length.toSystem("Imperial");
```

In this case, the names of the from-system and to-system may be omitted from the call to the `convert` method. This is useful if several unit conversions between the SI and Imperial systems are required.

Lines 33 and 34 convert pounds (in the US system) to kilograms (in the SI system) and kantars (in the Egyptian system). Kantars are used to measure cotton and correspond to hundredweight in the US system. Lines 41 and 42 convert liters to gallons in the US and UK systems. Note that a UK gallon is larger than a US gallon. Finally, lines 49 and 50 convert 37° Celsius (body temperature) to its Fahrenheit and Rankine equivalent. Fahrenheit and Rankine temperature are related by the equation °R = °F + 459.67. Unlike most unit conversions, temperature requires not only a value ($5/9$ in the case of Fahrenheit and Rankine) but an *offset*. In the case of Rankine, the offset is 459.67 (multiplied by $5/9$).

The output of the example program (which includes the unit conversions discussed above) is shown in Figure 4.7. Except for the precision, these are identical to those in Figures 4.5 and 4.6 for the C++ and C# versions of the example program. The precision is different since the default precision for each language is used when printing the results.

```
01 ////////////////////////////////////////////////////////////////////////////
02//
03 // File Example.java
04 //
05 // Program to demonstrate library usage.
06 //
07 ////////////////////////////////////////////////////////////////////////////
08 import UnitConversion.UnitConversions;
09 import UnitConversion.Converter;
10 import java.io.*;
11
12 public class Example
13 {
14    public static void main(String[] args)
15    {
16        System.out.println("Start Example");
17        UnitConversions unitConversions = UnitConversions.Instance();
18        Converter  length = unitConversions.converter("Length");
19        Converter  mass   = unitConversions.converter("Mass");
20        Converter  liquid = unitConversions.converter("Liquid");
21        Converter  temp   = unitConversions.converter("Temperature");
22
23        // Length conversion: convert kilometers to meters and miles.
24        double km = 3.5;
25        double m  = length.convert(km, "kilometer", "meter");
26        double mi = length.convert(km, "kilometer", "SI", "mile", "Imperial");
27        System.out.println("Length");
28        System.out.println(km + " kilometers = " + m  + " meters" );
29        System.out.println(km + " kilometers = " + mi + " miles");
30
31        // Mass conversion. Convert pounds to kilograms and Egyptian kantar.
32        double pounds = 100.0;
33        double kgm    = mass.convert(pounds, "pound", "US", "kilogram", "SI");
34        double kantar = mass.convert(pounds, "pound", "US", "kantar",   "Egyptian");
35        System.out.println("Mass");
36        System.out.println(pounds + " pounds = " + kgm    + " kilograms");
37        System.out.println(pounds + " pounds = " + kantar + " kantar");
38
39        // Liquid volume conversion: convert liters to gallons
40        double liters = 12.0;
41        double gUS    = liquid.convert(liters, "liter", "SI", "gallon", "US");
42        double gUK    = liquid.convert(liters, "liter", "SI", "gallon", "UK");
43        System.out.println("Liquid Volume");
44        System.out.println(liters + " liters = " + gUS + " gallons (US)");
45        System.out.println(liters + " liters = " + gUK + " gallons (UK)");
46
47        // Temperature Conversion: Celsius to Fahrenheit and Rankine
48        double celsius    = 37.0;
49        double fahrenheit = temp.convert(celsius, "Celsius", "INT", "Fahrenheit", "Imperial");
50        double rankine    = temp.convert(celsius, "Celsius", "INT", "Rankine",    "Imperial");
51        System.out.println("Temperature");
52        System.out.println(celsius + " Celsius = " + fahrenheit + " Fahrenheit");
53        System.out.println(celsius + " Celsius = " + rankine    + " Rankine");
54        System.out.println(" ");
55
56        System.out.println("End Example");
57    }
58
59 }
60 // EOF
```

Listing 4.3: *Java Example.* Java example showing unit conversion of length, mass, liquid volume, and temperature.

Python Example

Listing 4.4 shows an example of using the Unit Conversion Library in a simple Python script. The the Unit Conversion Library is placed at the top of the project directory hierarchy for the example and the `PYTHONPATH` set accordingly. Line 8 of the listing imports the `UnitConversions` class which contains all the units in the library. In C++, C#, and Java this class is a singleton; its constructor is private and it contains a static `Instance` method to obtain the single instance of the class. This is mimicked in Python by having a static `Instance` method included in the class. Of course, in Python the class constructor may be called, but this is discouraged for reasons outlined in in §6, **Software**.

Line 15 of the listing uses the static `Instance` method to obtain access to the Unit Conversion Library. This is immediately followed by calls to the class's `converter` method to obtain `Converters` for length, mass, liquid volume, and temperature. Each unit type has a converter for performing conversions on equivalent units. A list of units types can be found in the Appendix.

Lines 23 and 24 of the listing use the `convert` method to convert 3.5 kilometers to meters and miles. A unit is uniquely defined by its type, system, and name. The `convert` method requires the unit name and system to identify the unit. The unit's system is termed the from-system and its system after conversion is termed the to-system. The convert method requires the name of both the from-system and the to-system. Each `Converter` has a default from-system and to-system. The names of these systems may be obtained by calling the converter's `fromSystem` and `toSystem` methods. The defaults may be changed using these methods with the names of the new default systems.

For the case of converting kilometers to meters, both units are part of the SI system and since the length converter's default from-system and to-system is the SI system, the from and to system names may be omitted when calling the `convert` method, as shown on Line 23 of the listing. This is not the case for converting kilometers to miles since miles is part of the Imperial system of units. In this case the system names are required, as shown on line 24 of the listing. As an alternative, a call to the `toSystem` method could inserted between lines 23 and 24 to change the default to-system:

```
length.toSystem("Imperial");
```

In this case, the names of the from-system and to-system can be omitted from the call to the `convert` method. This is useful if multiple conversions between these systems is required.

Pounds is converted to kilograms and Egyptian kantars on lines 32 and 33 of the listing. Note the use of system names in the call to the `convert` method. Kantars are the US equivalent of hundredweight and are used to measure cotton. Lines 41 and 42 convert liters to US and UK gallons. US gallons are about 20% smaller than UK gallons.

Finally, lines 50 and 51 convert body temperature in Celsius to Fahrenheit and Rankine. The results of the example script (which contains additional conversions) is shown in Figure 4.8. A comparison of Figures 4.5-4.8 shows identical results expect for the output precision of the converted values. The default printing precision for each language was used.

```
01 ##############################################################################
02 #
03 # File Example.py
04 #
05 # Program to demonstrate library usage.
06 #
07 ##############################################################################
08 from UnitConversionLib.UnitConversion import UnitConversions
09 import os
10 import sys
11
12 def main():
13    print("Start Example")
14
15    unitConversions = UnitConversions.UnitConversions.Instance()
16    length = unitConversions.converter("Length")
17    mass   = unitConversions.converter("Mass")
18    liquid = unitConversions.converter("Liquid")
19    temp   = unitConversions.converter("Temperature")
20
21    # Length conversion: convert kilometers to meters and miles.
22    km = 3.5
23    m  = length.convert(km, "kilometer", "meter")
24    mi = length.convert(km, "kilometer", "SI", "mile", "Imperial")
25    print("Length")
26    print(str(km) + " kilometers = " + str(m)  + " meters" )
27    print(str(km) + " kilometers = " + str(mi) + " miles")
28    print(" ")
29
30    # Mass conversion. Convert pounds to kilograms and Egyptian kantar.
31    pounds = 100.0
32    kgm    = mass.convert(pounds, "pound", "US", "kilogram", "SI")
33    kantar = mass.convert(pounds, "pound", "US", "kantar",   "Egyptian")
34    print("Mass")
35    print(str(pounds) + " pounds = " + str(kgm)    + " kilograms")
36    print(str(pounds) + " pounds = " + str(kantar) + " kantar")
37    print(" ")
38
39    # Liquid volume conversion: convert liters to gallons
40    liters = 12.0
41    gUS    = liquid.convert(liters, "liter", "SI", "gallon", "US")
42    gUK    = liquid.convert(liters, "liter", "SI", "gallon", "UK")
43    print("Liquid Volume")
44    print(str(liters) + " liters = " + str(gUS) + " gallons (US)")
45    print(str(liters) + " liters = " + str(gUK) + " gallons (UK)")
46    print(" ")
47
48    # Temperature Conversion: Celsius to Fahrenheit and Rankine
49    celsius    = 37.0
50    fahrenheit = temp.convert(celsius, "Celsius", "INT", "Fahrenheit", "Imperial")
51    rankine    = temp.convert(celsius, "Celsius", "INT", "Rankine",    "Imperial")
52    print("Temperature")
53    print(str(celsius) + " Celsius = " + str(fahrenheit) + " Fahrenheit")
54    print(str(celsius) + " Celsius = " + str(rankine)    + " Rankine")
55
56    print("End Example")
57 # EOF
```

Listing 4.4: *Python Example.* Python example showing unit conversion of length, mass, liquid volume, and temperature.

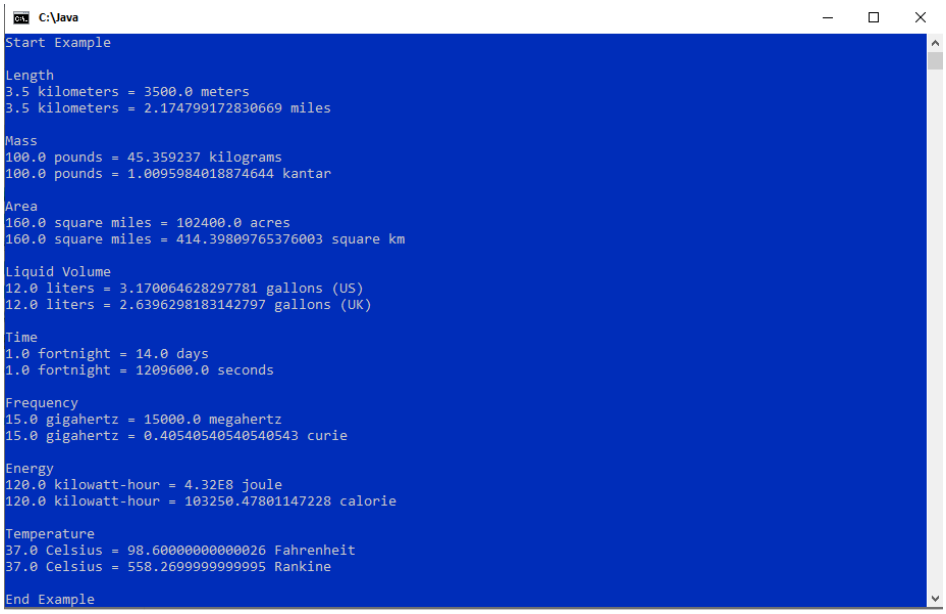

Figure 4.7: *Java Example Output.* Three and a half kilometers is converted to meters and miles, one hundred pounds to kilograms and Egyptian kantars, one hundred and sixty square miles to acres and square kilometers, twelve liters to gallons, a fortnight to days and seconds, fifteen gigahertz to megahertz and curies, one hundred twenty kilowatt-hours to joules and calories, and thirty-seven degrees Celsius to Fahrenheit and Rankine.

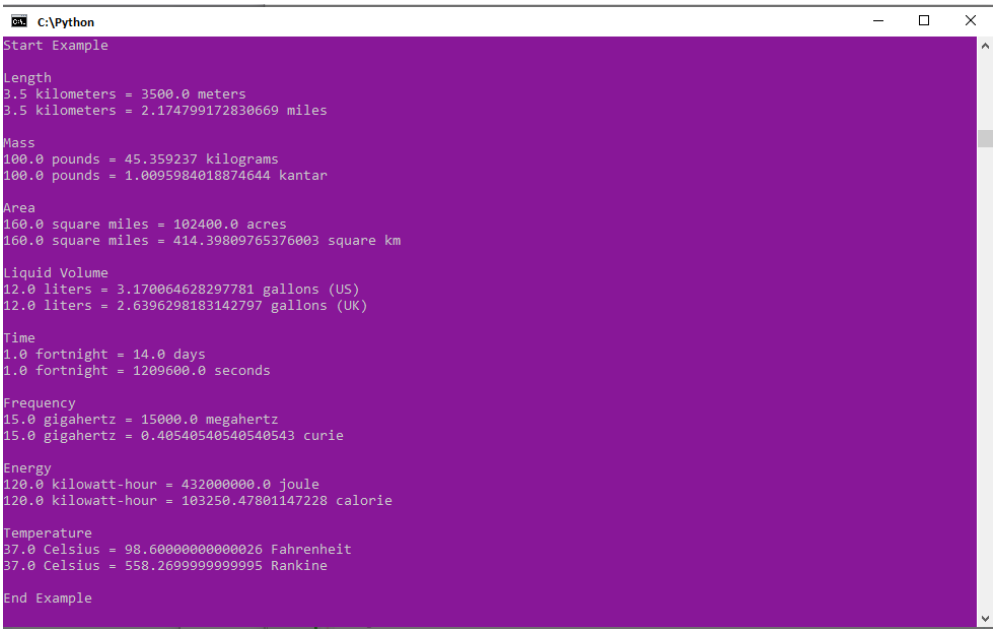

Figure 4.8: *Python Example Output.* Three and a half kilometers is converted to meters and miles, one hundred pounds to kilograms and Egyptian kantars, one hundred and sixty square miles to acres and square kilometers, twelve liters to gallons, a fortnight to days and seconds, fifteen gigahertz to megahertz and curies, one hundred twenty kilowatt-hours to joules and calories, and thirty-seven degrees Celsius to Fahrenheit and Rankine.

The information in this section should be sufficient to allow incorporation of the Unit Conversion Library into a project without reading the rest of the user's manual. A complete list of the units contained in the library to too long to include in this document. However, the `Converter` class contains `systemNames` and `unitNames` methods which, respectively, return a list of the system and unit names supported by the converter. These lists will, of course, be different for each `Converter` since they are type dependent.

Error Handling

The Unit Conversion Library does not throw errors keeping to the (admittedly arguable) philosophy that third part libraries should not disrupt a program's flow of processing. Instead, the `convert` method returns `NaN` (not a number) if an error is encountered. This usually means that the unit or system names supplied to the `convert` method could not be found. Spelling and capitalization of these names is important and must be exact. As such, the return value of the `convert` method can be checked using the `isError` method. This method takes the return value of the `convert` method and checks if it is `NaN`. If so, it returns `true`, otherwise it returns `false`.

In addition, the `Converter` class has a `valid` method that returns `true` if the converter contains at least one system of units containing one or more units. The following code fragment shows how these methods might be used to check for possible errors:

```
Converter temp = unitConversions.converter("Temperature");
if(temp.valid())
{
    double fahrenheit = temp.convert(celsius, "Celsius", "INT", "Fahrenheit", "Imperial");
    if(temp.isError(fahrenheit))
    {
       // Handle error…
    }
    else
    {
       // Proceed with processing…
    }
}
else
{
       // Handle error…
}
```

For more information on library organization, software classes, methods and other features see §6, **Software** and for more a more detailed discussion of unit conversion see §5, **Unit Conversion**. Also note that the Introduction contains definitions of terms used throughout this user's manual.

5.0 Unit Conversion

The **Introduction** contains definitions of terminology used throughout this user's manual and should be reviewed before reading this section. The word 'unit', or more precisely *unit of measurement*, is broadly defined as an agreed upon standard for measuring a physical quantity. Historically, units of measurement go back at least 5000 years (and almost certainly much further) with the Sumerians being among the first to develop a system of uniform weights and measures. Located in what is now Iran, they developed a base-60 or sexagesimal numerical system based on astronomical observations which is still used today for measuring angles and time. Other civilizations followed including the Greeks, Egyptians, and Romans. Units of measurement were often localized then superseded or 'standardized' by larger and more powerful regional authorities. By the 1400's however highly localized units of measure were common in European towns and cities. Each town had their own standards for length, mass and volume (*canonical units*) , that were often part some public building or statue where they were easily accessible. Units might have common names in a particular country but their values differed from region to region or even from town to town. According to Wikipedia [7]: "*It has been estimated that, on the eve of the Revolution, a quarter of a million different units of measure were in use in France.*" It is probably safe to assume a similar situation in other European countries and in other parts of the world.

Although the situation has improved greatly in the last 200 years or so, these ancient and historical units are still used or referenced and along with modern systems, this makes unit conversion both necessary and essential. Cardarelli [1, 2] has cataloged many of these units along with their *metrification* values helping facilitate unit conversion. However, a few points are with considering. Cardarelli [2] includes, for example, Universal Time Coordinated (UTC), as a unit of time but does not (understandably) provide a conversion factor (to seconds). Instead, he writes:

> "*Universal Time Coordinated replaced GMT on January 1st, 1972. It corresponds to the TAI fitted to one second to ensure approximate time scale concordance with UT_1. Universal Time Coordinated is based on emitted coordinated time rate signals and standard frequencies. The international abbreviation UTC is employed in all languages. UTC = TAI – 10[1] seconds (SI).*"

An examination of the unit entry for TAI (International Atomic Time) yields that it's a time unit but again there is no conversion factor. Instead, a note is again provided:

> "*The TAI corresponds to a time scale established by the Bureau International de l'Heure (BIH) given by atomic clocks in several locations through the world. The international abbreviation TAI is employed in all languages.*"

This is all well and good except that is leaves the question: How does once convert to and from these time 'units'? In the broadest sense of the definition of *unit of measurement,* both UTC and TAI qualify as units since they measure a physical quantity, time, and a such Cardarelli is correct in including them in his catalog. However, converting between them is not as simple as multiplying by a conversion factor (value). Time is a complicated physical quantity as shown by Einstein and unlike other physical quantities like length or volume or mass, time is dynamic not static. As noted by Jespersen and Fitz-Randolph [8]:

[1] The current time difference (as of this writing) is 37 seconds.

"We can see distance and feel weight and temperature, but we cannot apprehend time by any of the physical senses…Time 'passes', and it moves in only one direction…'Now' is constantly changing. We can buy a good meter stick, or a one-gram weight, or even a thermometer, put it away in a drawer or cabinet, and use it whenever we wish. We can forget it between uses – for a day or a week or 10 years – and find it as useful when we bring it out as when we put it away. But a "clock" – the 'measuring stick' for time – is useful only if it is kept 'running'."

They go on to note that: "*At least part of the trouble in agreeing on what time it is lies in the use of a single word* time *to denote two distinct concepts. The first is* date *or* when *an event happens. The other is* time interval*, or the 'length' of time between two events*."

A similar problem exists with the word 'unit'. It is possible to convert between UTC and TAI and to compute the number of seconds, say, between midnight June 15, 1215 (King John fixed his seal to the Magna Carta) and midnight July 20, 1969 (Neil Armstrong and Buzz Aldrin walk on the moon). However, this is a non-trivial calculation. Once cannot simply subtract 1215 from 1969 and multiply by 365, add 35 (the number of days between June 15 and July 20) and then multiply by 86,400 (number of seconds in a day). The result 23,781,168,000 is wrong. The actual number of seconds is 23,796,374,400, a difference of 15,206,400 seconds or 176 days which results from not taking leap years (and seconds) into account as well as the change from the Julian to Gregorian calendars in 1582.

The problem is that UTC and TAI (and other such systems) are more than simple 'units'. They are in reality *reference* or *coordinate* systems tied to the earth and the 'conversion' from one to another or even the computation of time differences within the same system is a non-linear transformation. This poses a fundamental problem for unit conversion. Consider the problem of transforming length from say feet to meters. Since one foot is equal (by definition) to exactly 0.3048 meters we can write the following linear equation:

$$meters = 0.3048 \times feet$$

Now suppose we have a system whose length unit is called a knee and this unit transforms to meters as follows:

$$meters = a \times (knee)^2$$

where a is some fixed constant. Now consider two rulers: one that measures length in feet and the other in knees. Both rulers have a scale with two marks denoting 1 and 2 feet on the first ruler and 1 and 2 knees on the second, as shown in Figure 5.1. Using the equations above we can convert these scales to meters and add a second meter scale to both rulers as shown in the figure. On the foot ruler, the distance between the left edge and the first mark is one foot or 0.3048 meters and the distance between the first mark and the second mark (right edge) is also one foot or 0.3048 meters.

Now consider the knee ruler. The distance between the left edge and the first mark is one knee or a meters. The distance between the first mark and the second mark (right edge) is also one knee. However, the distance between these two marks on the meter scale is $3a$ meters. Clearly something is amiss here. We expect that for units of length, the *distance* (length between two points) be *invariant* with respect to the conversion. That is the numerical value of the distance can change when converted

but that equal distances must have equal numerical values after being converted. This is clearly true for the feet to meter conversion but *not* for the knee to meter conversion. We expect invariance for all *proper* unit conversions regardless of whether the unit measures length, area, mass, volume, electric field strength or time.

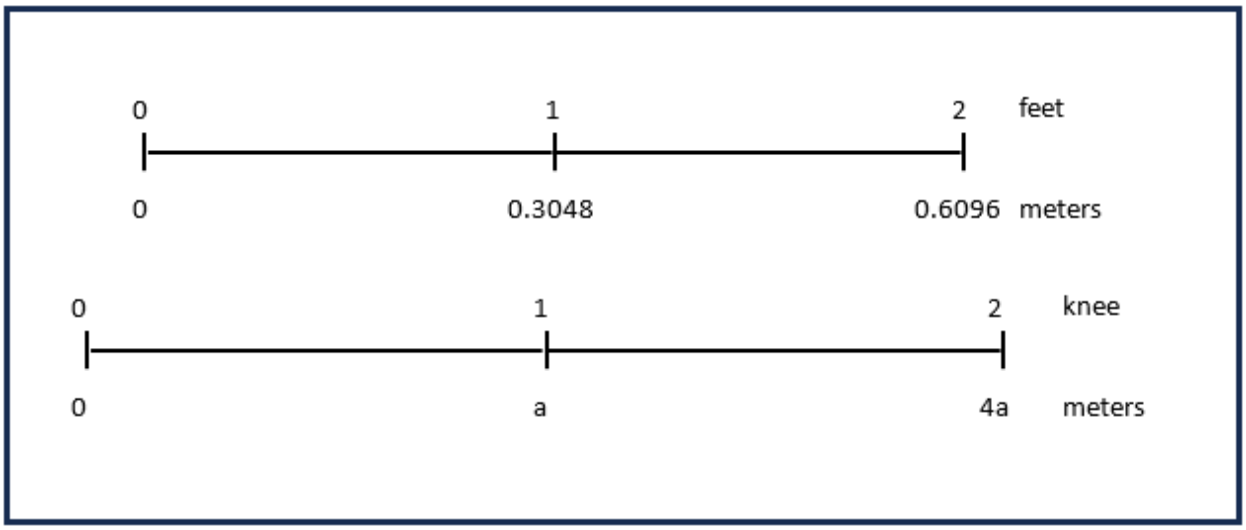

Figure 5.1: *Non-Linear Unit Conversion Example.* Since the knee to meters conversion is non-linear distance is not invariant and therefore this conversion is not proper. It is instead a transformation, not a conversion, unlike the feet to meters conversion which is linear and therefore a proper conversion of units.

The problem of course, is that the knee to meter 'conversion' is non-linear and as such distance is 'stretched' when 'converted'. This true of all non-linear transformations and is clearly unacceptable when applied to unit conversions. We therefore make the following definitions:

> A *unit conversion* is a linear (affine) transformation between equivalent units in different systems or within the same system.
>
> A unit is *convertible* if it can be transformed to an equivalent unit using a unit conversion.
>
> A *proper unit of measurement* (or proper unit) is one that is convertible.

This means that UTC and TAI as well as decibel units and the like are not proper units since they are not *convertible*; rather they are *transformable*, meaning for example that time in TAI can be transformed into an equivalent UTC time but that this transformation is not a *unit conversion*, as defined above. The Unit Conversion Library only contains proper units; units such as UTC and TAI are therefore specifically excluded. This however does not mean that time is exclude from the library.

Time, as Jespersen and Fitz-Randolph [8] state, refers to both date and an interval or duration. Dates are not proper units and as such are not convertible. Durations however are proper units and are convertible. The time unit *second* is part of the SI system of units and forms the basis for measuring time intervals as well at computing dates. The time units minutes, hours, days, and years are well defined multiples of the second and are therefore proper units. There are however different 'flavors' of these units such as the sideral year, the Gaussian month, and the anomalistic day. The 'standard' or fixed

definition of a day is 86,400 seconds and a month is 30 days and a year 365 days. The anomalistic day , Gaussian month, and sideral year are tied to the earth's rotation on its axis and its orbit around the sun. Since the earth's rotation about its axis is slowing due to tidal effects and other phenomena these time periods are not strictly fixed and as such are not proper units. However, the variation in their values is small and they have been included in the library with the caution that their accuracy is probably not more than a second. If sub-second accuracy is required, then the Unit Conversion Library should not be used. C++, C#, Java, and Python all come with built-in functions that handle dates and these should be used for calculations such as computing the number of seconds between the signing of the Maga Carta and men walking on the moon.

As noted in the **introduction**, the Unit Conversion Library uses double precision arithmetic for all conversion calculations. Unit values and offsets are stored to 15 digits base-10 providing maximum precision for conversion. However, precision does not directly equate to *accuracy*. For units such as feet which are *defined* has having exact values, the accuracy of the conversion equates with the precision of the calculation. For other units, especially ancient and historical units, this is not the case. Take perhaps the most well known ancient unit of measurement, the *cubit*. According to Wikipedia [9]:

"The cubit is an ancient unit of length based on the distance from the elbow to the tip of the middle finger. It was primarily associated with the Sumerians, Egyptians, and Israelites. The term cubit is found in the Bible regarding Noah's Ark, the Ark of the Covenant, the Tabernacle, and Solomon's Temple...Cubits of various lengths were employed in many parts of the world in antiquity, during the Middle Ages and as recently as early modern times...The ancient Egyptian royal cubit is the earliest attested standard measure. Cubit rods were used for the measurement of length. A number of these rods have survived…Fourteen such rods, including one double cubit rod, were described and compared by Lepsius in 1865. These cubit rods range from 523.5 to 529.2 mm in length..."

Cardarelli [1] gives a value of 52.35 cm for the royal derah (royal cubit) placing it at the lower end of the known range of cubit lengths. Figure 5.2 shows an example of an ancient cubit rod from the 18th dynasty. It should not be surprising that the length of a cubit has varied over time. The length of the modern meter for example has changed (albeit slightly) several times over the last 230 years. According to Wikipedia [10]:

"*The meter was originally defined in 1791 by the French National Assembly as one ten-millionth of the distance from the equator to the North Pole along a great circle...in 1799, the meter was redefined in terms of a prototype meter bar, the bar used was changed in 1889, and in 1960 the meter was redefined in terms of a certain number of wavelengths of a certain emission line of krypton-86. The current definition was adopted in 1983 and modified slightly in 2002 to clarify that the meter is a measure of proper length. From 1983 until 2019, the meter was formally defined as the length of the path travelled by light in vacuum in 1/299792458 of a second. After the 2019 redefinition of the SI base units, this definition was rephrased to include the definition of a second in terms of the cesium frequency* Δv_{Cs}*. This series of amendments did not alter the size of the meter significantly… a change of 0.022% from the original value*…"

This change, about 0.22 millimeters, is smaller than the known 5.7-millimeter variation in the cubit, but not dramatically so, especially considering the large changes in technology over the last 3,000 plus years. Unlike the meter however the cubit's length depends on archeological discoveries and historical records;

there is no fixed exact definition. The same holds true for other (all) ancient units and most historical units.

Figure 5.2: *Egyptian Cubit Rod.* This cubit rod is from rom the tomb of Maya, the treasurer of the 18th dynasty pharaoh Tutankhamun. It measures 52.3 centimeters long.

This raises two issues with accuracy. The first is one of *knowledge* of the values of these units and the second is their *variation* over time and location. In the case of the cubit an actual physical example exists from ancient times that can be examined and measured. The values of these physical examples have probably changed at least slightly over the centuries, as for example wood shrinks as it ages. In addition, the ancient Egyptians lacked the technology (and the need) to accurately produce a rod or bar whose length was accurate to with a wavelength of light or 1/299792458 of a second, as is the case for the modern meter. The result is, that knowledge limits the accuracy of these units and as such, one cannot expect the same high degree of accuracy in ancient and historical units as in modern units of measurement.

The second issue of variation is perhaps more difficult to handle on a practical level and applies to all units, ancient, historical, and modern. As noted above there were over a quarter of a million units in use in France alone before the revolution (Wikipedia [7]). No doubt many of these units went by the same name, but their values changed based on location. One has to assume that the same variation with location occurred in other European countries as well as other parts of the world. Adding to the problem is that the value of these units almost certainly varied over time as well. Even if all these variations could somehow be cataloged, the resulting collection would be enormous and finding the value of a specific unit would require specifying its physical location and date of usage precisely. Modern information systems could obviously handle storage and access, but determining which ‘version’ of a unit was desired would require an extensive knowledge of both the geography and history of the desired unit.

The net result of this is that the Unit Conversion Library provides precise results when converting units. Precise means 15 digits, base-10, the maximum available using double precision arithmetic on a modern computer. Accuracy however depends on the unit. For modern units such as the meter, accuracy is limited only by the precision of the input unit (unit to be converted) and the machine (computer). For ancient and historical units, accuracy is limited by knowledge of the unit’s historical value and its variation over time and location. Determining this accuracy is beyond the scope of the Unit Conversion Library and this burden is placed on the user to ascertain, if required.

Cardarelli [1, 2] lists unit values to 10 digits, base-10 of precision and other sources list values to varying degrees of precision. Most units however are derived from other units, usually as multiples of these units. For example, the inch is $1/12$ of a foot and quart is $1/4$ of a gallon. In each system of units there are a small subset of units from which all the other units in the system are derived. These units are termed *system units* and form a basis for that system of units. By way of example, the foot, square-foot, pound, liquid and dry gallon (yes, they are different) are the (canonical) system units for the US system of

measurement. The seven units: meters, kilograms, seconds, kelvin, mole, ampere, and candela constitute the system units for the SI system. Of the 5,000 plus units in the Unit Conversion Library most are derived from a smaller set of system units.

The choice of system units is somewhat arbitrary (e.g. foot instead of inches for length in the US system) and probably historical or by common convention, tradition, or informal agreement. However, system units greatly simplify the problem of precision since specifying these units to maximum precision and then deriving the remaining units in the system from them ensures uniformity of the precision of their values. This is the approach taken in the Unit Conversion Library. All units (except system units) are derived; system units are defined. The **Appendix** contains a table (Table 9.7) giving the system units for all the major systems in the library. This table has about 90 systems with system unit values for each of its canonical units.

In addition to system units, some units require additional parameters to derive their values. For example, the Bohr Radius uses Plank's constant, the elementary charge, the electron rest mass, and the permittivity of free space to compute its value. Plank's constant and the elementary charge are *physical constants* (**Appendix**, Table 9.4) with defined values. The electron rest mass, and the permittivity of free space are *physical measurements* (**Appendix**, Table 9.6). Other units such as the minute and hour are *physical definitions* (**Appendix**, Table 9.5). The precision the entries in Table 9.6 varies with the quantity being measured but typically does not exceed 10 digits base-10.

Using the values in Tables 9.4-7 allows any unit in the library to be derived as either a multiple of its system units or a combination of physical constants, definitions, and measurements. Obviously, the accuracy of any unit derived using a physical measurement cannot exceed the accuracy of the measurements used. Units derived using physical constants and definition can have greater accuracy since the values of both are exact. The accuracy of such units is therefore determined by the accuracy of their corresponding system units.

6.0 Software

The Unit conversion Library consist of four components: an Excel Spreadsheet containing the unit values, the underlaying VBA code used to generate the unit classes, the generated code, and the library classes. Every unit in the library has an entry in the spreadsheet where its value, offset, and type (dimensions) are stored. The VBA code embedded in the spreadsheet reads these unit parameters and generates code in the four supported languages (C++, C#, Java, and Python). This code consists of classes that inherit from the library classes and contain constructors with the units. The generated code is the 'data' part of the library. The library classes contain the code for storing, organizing, and accessing this data. This is the user interface to the library. Figure 6.1 shows these components and the flow from spreadsheet to library.

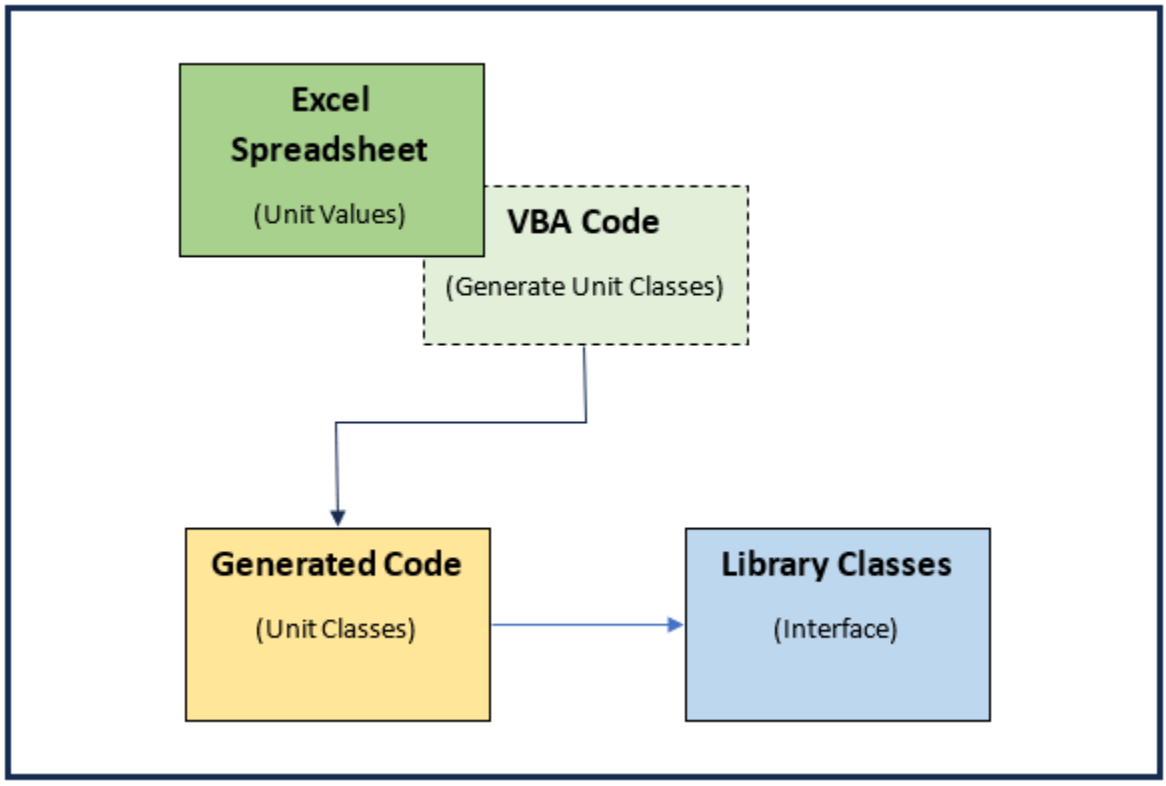

Figure 6.1: *Library Components.* The Unit Conversion Library consists of an Excel spreadsheet containing the unit information, VBA code to generate classes containing the spreadsheet data, the generated code with the unit data, and the library classes which provide an interface to the units stored in the generated code.

Excel Spreadsheet

The Excel spreadsheet acts as a database for the unit data. The unit data consists of the unit name, name abbreviation(s), system, type (dimensions), and SI units. The SI units are given in base form. That is for example, force is expressed as $kg.m.s^{-2}$ not as Newtons since Newtons is a derived unit. The notation used to express dimension and units is:

```
u1#.u2#....un#/U1#.U2#....Um#
```

where u1, u2,…,un, U1, U2,…Um are SI base units (see Table 1.1), and # is the unit exponent (if one, omitted). Units are separated by a period (.) with units to the left of the forward slash (/) in the numerator (positive exponent) and units to the right of the forward slash in the denominator. The unit for force is therefore written as kg.m/s2. The same notation is used for type (dimension). Again, for force this is: M.L/T2.

Each unit in the spreadsheet is calculated from a set of base or *system units*. These calculations are performed in double precision arithmetic and values are stored to 15 digits base-10 in scientific notation for maximum precision[2]. The units in the spreadsheet are organized by category with one worksheet (tab) per category. Figure 6.2 shows the layout of the workbook.

The tabs or unit categories correspond roughly (but not exactly) to the unit type. For example, the Angle worksheet contains units that have a dimension of radian (angle) as part of their unit composition. These include plane angles, solid angles, angular velocity, and angular acceleration. The cationization is an arguably logical method of organizing the unit data. However, within each category units are grouped by type and it is only this final typing that is visible in the software interface.

Two worksheets are of note. The first is the Generate worksheet shown in Figure 6.4. This worksheet contains some basic instructions and the `Generate` button which is linked to the VBA code used for generating the unit classes. These classes contain the unit data parsed and formatted from the workbook. Pressing this button calls the VBA code which generates these files. The generated unit class files are placed in a directory named `Output` at the same level as the Excel spreadsheet. This folder must exist before using the generate button. Any files in the `Output` directory are overwritten *without* warning. There is no need to regenerate the unit data classes unless the Excel spreadsheet is modified by adding, removing, or editing the unit data. The cell next to the `Generate` button displays the status of the running VBA code. It reads `Done` when all the code has been generated. The cells above the `Generate` button labeled `Date` and `Version` are the date and versions that appear in the generated code. The unit version information can be checked against the software version when using the library (see below).

The second worksheet worth noting is the `Systems` worksheet. This worksheet contains all the *system units*. System units are a canonical set of units from which length, mass, area, and volume are derived for a particular unit system. For example, the US system uses feet, pounds, square feet, and gallons as its system units. Other units such as inches, ounces, acres, and quarts (for example) are multiples of these units. This choice of system units is arguably arbitrary and is typically historical, customary, or by common convention or agreement. These units are important since most of the units in the workbook are derived or calculated using them. In addition, and without exception every unit value and offset in the workbook is calculated.

Some units require various constants in order to calculate their value. These are units such as the Bohr radius and nuclear magneton. The workbook contains the following worksheets which define these constants: Physical Constants, Physical Definitions, and Physical measurements (see Tables 9.4-6 in the **Appendix**). Physical constants are defined and have exact values. These are Planck's constant, the speed of light, the elementary charge, Boltzmann's constant, and Avogadro's constant. Physical definitions are again exact and include units such as the minute, hour, carat, gram, and curie. Physical measurements

[2] See §5 for a discussion on precision versus accuracy.

are quantities such as the electron rest mass, Dalton, and density of water at various temperatures. Many of these measurements have limited precision, often to 10 decimal places. Other worksheets include the derived units (coulomb, farad, newton, watt, etc.), SI units (see Table 1.1) and the WGS84 model. The `Binary`, and `Prefix` worksheets are for reference and the Units worksheet for verification (see §8**, Unit Naming Conventions**).

Each unit worksheet contains the unit data for all the unit types in that category. Nine pieces of information are captured for each unit. These are the unit's name (which must be unique for its type and system), any abbreviations for the unit, the unit's type in words, the units dimensions, the unit's value, the base for the unit, the unit's SI components, and the system containing the unit. The base of the unit is the system units and physical constants, measurements, and definitions used to calculate the unit's value. Figure 6.2 shows an example from the `Angle` worksheet.

The `Systems` worksheet contains the system units and has a different layout. This worksheet is organized by system and contains five blocks of data for each system. Each block consists of four columns giving the system unit name, value, base, and SI unit. The five blocks correspond to length, mass, area, liquid volume, and dry volume. Each system typically as at least one unit in each of these five blocks. System units are treated as constants in the Unit Conversion Library (see below). Care should be exercised in editing this worksheet as changes here affect the values of units in the other worksheets.

The `Temperature` worksheet contains an additional column for the offset value. Temperature is the only unit type that has a non-zero offset value. The offset value is located between the unit value and its base.

Generated Code

The Excel workbook contains VBA code for generating classes in the four supported languages that contain the unit data in the worksheets. These are termed *unit classes* and they consist of a private constructor and a static instance method i.e. singletons. The private constructors contain the units; one unit class is created for each category (worksheet). The unit classes in the generated code, for the most part, all follow the same general format:

- Imports or includes.
- Class declaration based on language.
- A public static instance method.
- A private constructor.

Listing 6.1 shows an example for the `Angle` unit class. As this code is generated it should not be edited. Modifications should be made via the spreadsheet and underlying VBA ode. For Python, the concept of 'private' does not fully exist, so of course, unit class constructors are accessible, however the static instance method is included and this is the recommend method of accessing the unit data within the class.

As shown in Listing 6.1, the `Angle` class inherits from the library class named `SingleSystem`. The `SingleSystem` class contains the accessor and other functions for accessing and manipulating the unit data (see below). Most of the generated unit classes inherit from this class. The most notable exception is the canonical units (again, see below) and the system units and constants.

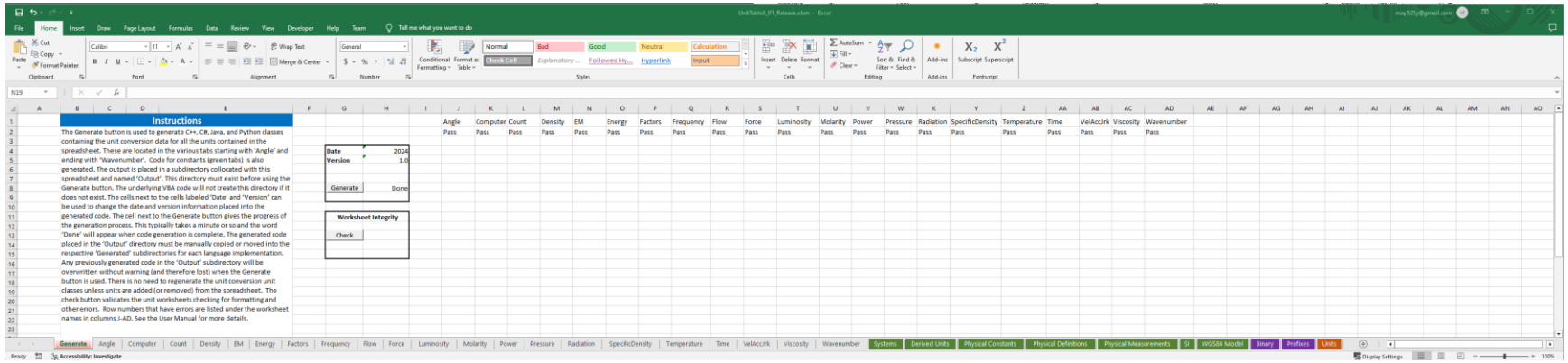

Figure 6.2: *13Excel Spreadsheet.* The spreadsheet is organized by unit category with one worksheet (tab) per category. The green tabs are constants, purple are informational. The red tab (Generate) is used to invoke the VBA code which generates files containing the unit class code (one per each language).

	A	B	C	D	E	F	G	H
1	Unit Name	Symbol	Physical Quantity	Dimensions	Conversion Factor	Base	Unit	System
2	angle-droit	D	angle, plane angle	α	1.570796326794900E+00	3.141592653589790E+00	rad	French
3	centrad	-	angle, plane angle	α	1.000000000000000E-02	1.000000000000000E+00	rad	Angle
4	circumference	-	angle, plane angle	α	6.283185307179590E+00	6.283185307179590E+00	rad	INT
5	degree	-	angle, plane angle	α	1.745329251994330E-02	1.745329251994330E-02	rad	INT
6	degree	-	angle, plane angle	α	1.745329251994330E-02	1.745329251994330E-02	rad	Angle
7	dekan	-	angle, plane angle	α	1.745329251994330E-01	1.745329251994330E-01	rad	Angle
8	dimigrade	dmgr	angle, plane angle	α	1.570796326794900E-06	1.570796326794900E-02	rad	French
9	gon	-	angle, plane angle	α	1.570796326794900E-02	1.570796326794900E-02	rad	INT
10	gon	g	angle, plane angle	α	1.570796326794900E-02	1.570796326794900E-02	rad	French
11	grade	gr	angle, plane angle	α	1.570796326794900E-02	1.570796326794900E-02	rad	French
12	hour-of-angle	hoa	angle, plane angle	α	2.617993877991490E-01	2.617993877991490E-01	rad	Angle
13	house	-	angle, plane angle	α	5.235987755982990E-01	5.235987755982990E-01	rad	Angle
14	mas	-	angle, plane angle	α	4.848136811095360E-09	4.848136811095360E-09	rad	@
15	mil	-	angle, plane angle	α	1.022653858590430E-03	1.022653858590430E-03	rad	Angle
16	milangle	-	angle, plane angle	α	9.817477042468100E-04	9.817477042468100E-04	rad	INT
17	millieme	-	angle, plane angle	α	9.817477042468100E-04	9.817477042468100E-04	rad	INT
18	millieme	-	angle, plane angle	α	9.973310011396170E-04	9.973310011396170E-04	rad	Russian
19	millieme	-	angle, plane angle	α	1.570796326794900E-03	1.570796326794900E-03	rad	US
20	millieme	m_s	angle, plane angle	α	9.817477042468100E-04	9.817477042468100E-04	rad	French

Figure 6.3: *Angle Worksheet.* Each unit worksheet contains nine columns corresponding to the unit name (column A), symbol (or abbreviation(s), column B), type (in words, column C), dimension (column D), value(conversion factor, column E), system unit (base, column F), SI component units (column G), and system (column H).

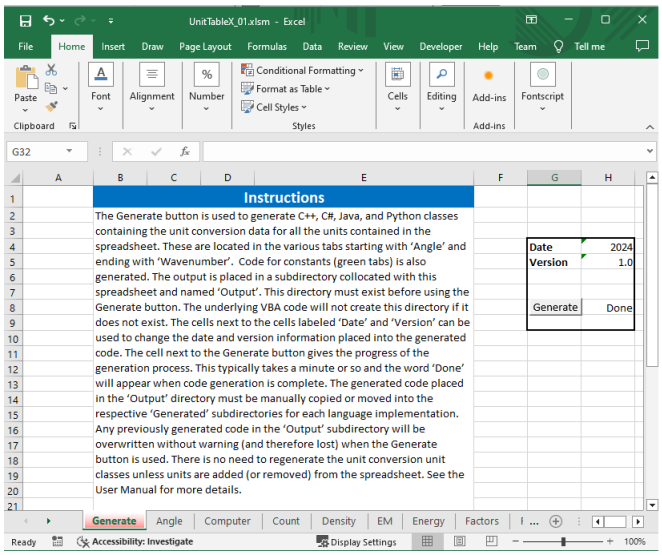

Figure 6.4: *Generate Worksheet.* This worksheet contains the `Generate` button which is linked to the VBA code that generates the unit classes which form the data portion of the Unit Conversion Library.

```
01 #pragma once
02
03 #include "SingleSystem.hpp"
04
05 class Angle : public SingleSystem
06 {
07 public:
08    static Angle& Instance(void);
09
10 private:
11    Angle(void);
12 };
```

Listing 6.1: *Generated Code Example (C++ header).* The generated code consists of import or includes (language dependent), class declaration, a public static `Instance` method, and a private constructor.

The generated code is placed in the `Generated` folders for each supported language in the `UnitConversionLibrary` folder. As noted above these classes should not be modified directly; the Excel spreadsheet and VBA code are the recommended method for modifying these classes.

VBA Code

The VBA code is part of the Excel spreadsheet that contains the unit data. This code goes though the worksheets in the workbook line by line and column by column and extracts the unit data and formats it into the unit classes. Each method in the VBA code contains a header with brief documentation. The spreadsheet is an `xlsm` file meaning that it contains executable code that is not part of the Excel application as delivered by Microsoft. As such, Windows may issue a warning or open this file in 'protected' mode. To run the VBA code (`Generate` button, see above) any such 'protection' of the spreadsheet file must be overridden. *It is strongly suggested that end users perform a virus scan on this file before opening it or lifting any protections imposed by Windows*.

The VBA code is divided into six code generators that correspond to the format and contents of the worksheets. These are listed in Table 6.1. The first, `WriteIntUnit` is used for integer units such as counts and computer units. Most unit values are double precision numbers except for counts and computer units such as bytes which are integer units.

Table 6.1: VBA Code Generators

Subroutine	Generator	Description
`WriteIntUnit`	`WriteAIntUnit`	Write integer units (Count and Computer).
`WriteTemp`	`WriteATemp`	Write temperature
`WriteSingleUnit`	`WriteASingleUnit`	Write Angle, Density, EM, and other single units.
`WriteCanonicalUnit`	`WriteACanonicalUnit`	Write canonical units (Factors worksheet).
`WriteConstantUnit`	`WriteAConstantUnit`	Write physical constants and SI units.
`WriteSystemUnits`	`WriteASystemUnits`	Write system units.

The `WriteTemp` generator is used for temperature units which contain an offset factor as part of their conversion. The majority of the generated unit classes inherit from the `SingleSystem` class and the `WriteSingleUnit` subroutine generates the code for these classes. Canonical units which comprise the majority of units in the library inherit from the `CanonicalSystem` class (see below) and this code is generated by the `WriteCanonicalUnit` subroutine. Physical constants and SI units classes are generated by the `WriteConstantUnit` subroutine. Finally, the system units are generated by the `WriteSystemUnits` subroutine. Each of these subroutines calls a generator subroutine or function which parses the relevant worksheet(s) and generates the unit classes. The generated code for each language is placed in the `Output` folder which is located at the same level as the Excel spreadsheet. The Python script `Distrubute.py` can be used to place this code into the `UnintConversionLibrary` folder. On Windows double clicking on the `RunDistribute.bat` file executes this script. Each code generator is run five times (once for C++ headers files, and once again for each supported language). These generators follow the same sequence of processing as shown in Figure 6.5.

Processing begins by opening the worksheet containing the unit data and opening an output file for writing the generated code. The worksheet is parsed by row and then column (double for loop). The cell data from each row and column is read and then written to the output file. The format of the output is based on the file extension and this extension determines the flow of the lower-level processing used to write the unit code. Each language has its syntactical idiosyncrasies and as a result each write requires calling a formatter specific to the language being written.

The generated unit code can be called directly from the library but this is not recommended. These classes are organized by category not type and as such type, system, and name information must be supplied for each conversion. Instead, the interface code (see below) should be used. The same holds true for the physical constants, measurements and definitions as well as the SI and system units. Using the interface is the preferred method to accessing these units.

Although the VBA code is available for inspection, it is not recommended that this code be altered. It is merely a means for transforming the data in the Excel spreadsheet into language specific classes. Modifying the VBA code can result in incompatibilities with the underlying interface classes leading to compiling errors (for C++, C#, and Java) and execution errors for Python.

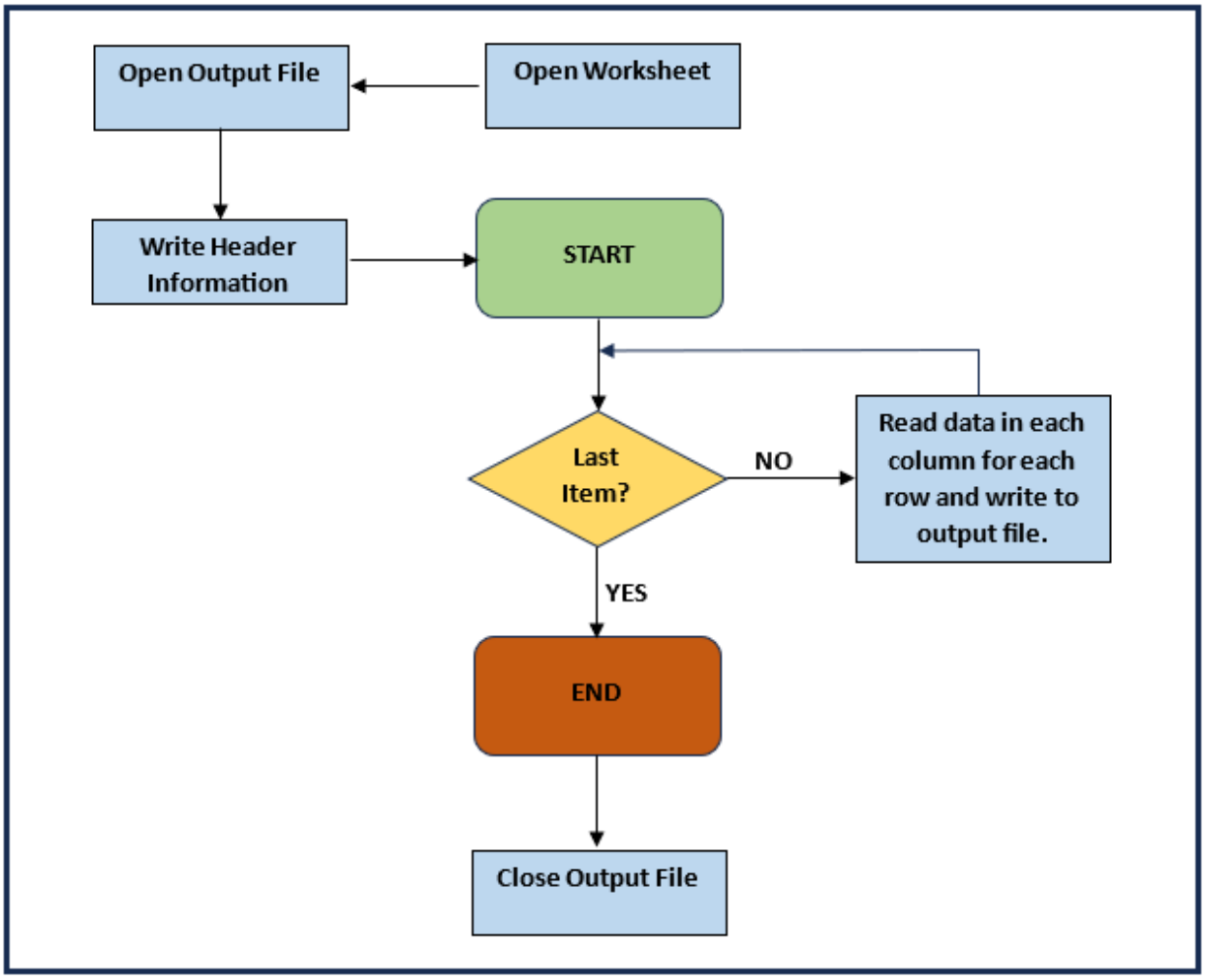

Figure 6.5: *VBA Worksheet Processing.* Processing begins by opening the worksheet with the unit data and an output file for writing the generated code. The worksheet data is parsed row wise and then column wise, extracting the unit data from each cell and writing it to the output file. When processing is complete the output file is closed.

Library Classes

There are fifteen classes that comprise the library of which three constitute the user interface. The remaining twelve provide storage for the unit data and lower-level access to this data. At the lowest level is the `UBASE` class which stores a single unit of measurement. This class has a public constructor and accessor methods to retrieve the type, system, value and other unit information. There are no methods for modifying, adding, or removing data making the class immutable. In essence the `UBASE` class contains a single quantum of information for a unit. The cells in a row of the Excell spreadsheet directly map one-to-one onto an instance of this class. Table 6.2 shows the accessor methods.

A unit's value and offset are stored in the `Value` class. Because counts and computer 'units' are integers, a generic unit value can be either a double or an integer. The `Value` class allows either to be stored without consuming extra memory. The `UBASE` class stores its unit values and offsets as instance of the `Value` class.

The UBASE class provides the lowest level of storage for a single unit. Unit conversion requires that units must be of the same type. As such, organizing units by type facilitates the conversion of one unit to another. The next step up the class hierarchy is the TypeGroup class. A *type group* is a collection of related units (UBASE objects) having the same type and belonging to a system or some other related collection of units.

Table 6.2: UBASE Accessor Methods

Method	Description
name	Returns the unit names as a string. This name can be used to access the unit from other container classes.
offset	Non-zero only for temperature.
system	Returns the name of the system the unit is part of as a string.
toString	Returns a string representation of the unit data in the class.
type	Returns the unit type as a string. Note this is the unit dimensions.
unit	Returns the SI components of the unit as a string.
valid	Returns true if the class contains valid unit data, false otherwise.
value	Returns the unit value in SI units. This is the conversion factor to SI units.
version	Return unit version information as a string.

Programmatically a type group consists of one or UBASE objects all having the same type. Examples include angle, length, mass, and volume. These units may or may not belong to the same system but since they are of the same type, conversions between them are possible. A system of units consists of one or more type groups which are collected into a BaseSystem. Ancient and historical systems consist of four type groups for units of length, weight, area, and volume. Figure 6.6 shows this hierarchy of classes. This is implemented using maps and dictionaries in the four supported languages. Thus, a type group contains a map or dictionary of UBASE objects and a BaseSystem contains a map or dictionary of TypeGroup objects.

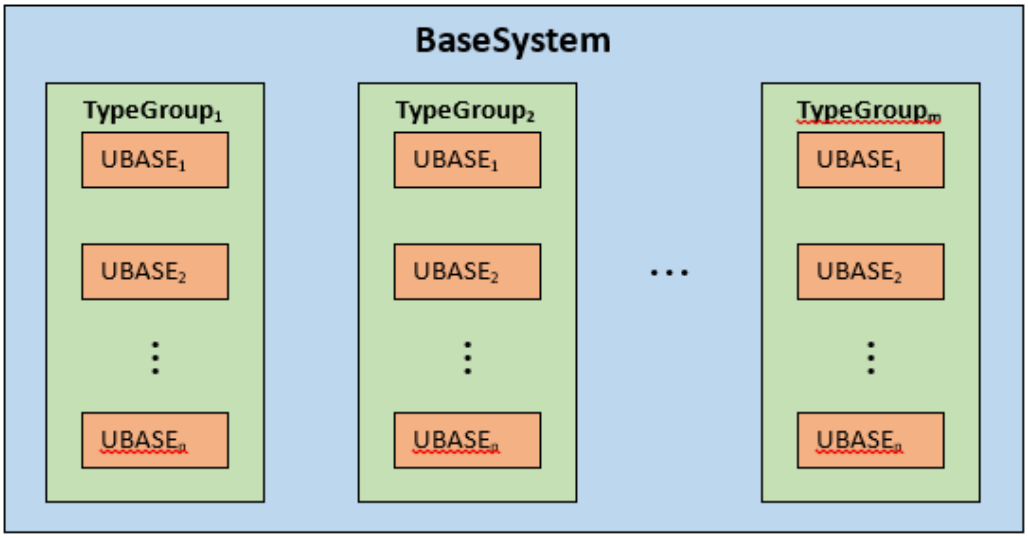

Figure 6.6: *17Low-Level Container Classes.* A BaseSystem consists of one or more TypeGroups. Each TypeGroup contains one or more UBASE objects. The UBASE class contains information for a single unit of measurement.

Since units are organized by type and then placed into categories of similar or related types in the Excel spreadsheet, another level of organization is required programmatically. For ancient and historical units which consist almost exclusively of canonical units, the `CanonicalSystem` class is used as their parent class. This class employes a map (or dictionary, depending on the language) of `BaseSystem` classes for organizing these units. In fact, even the canonical portion of modern units are included here. Each `CanonicalSystem` contains multiple `BaseSystem` classes (one per system) with each `BaseSystem` having five `TypeGroup` objects (length, area, mass, liquid volume, and dry volume). Relative to the Excel spreadsheet, the units in the `Factors` worksheet (which contains the ancient and historical canonical units) are placed in an instance of the `CanonicalSystem` class.

The remaining units (the other worksheets) require a `BaseSystem` containing only a single `TypeGroup`. For these units the `SingleSystem` class is used. Since the worksheets are organized by category, not type *per se*, a `SingleSystem` object consists of one or more `BaseSystem` objects each have a single `TypeGroup`. Figure 6.7 show the container relationship of these classes. The `SingleSystem` and `CanonicalSystem` classes are termed *foundation classes* as they serve as the base classes or foundation of all the generated units classes. These two classes provide the storage, access, and manipulation methods for the all unit classes.

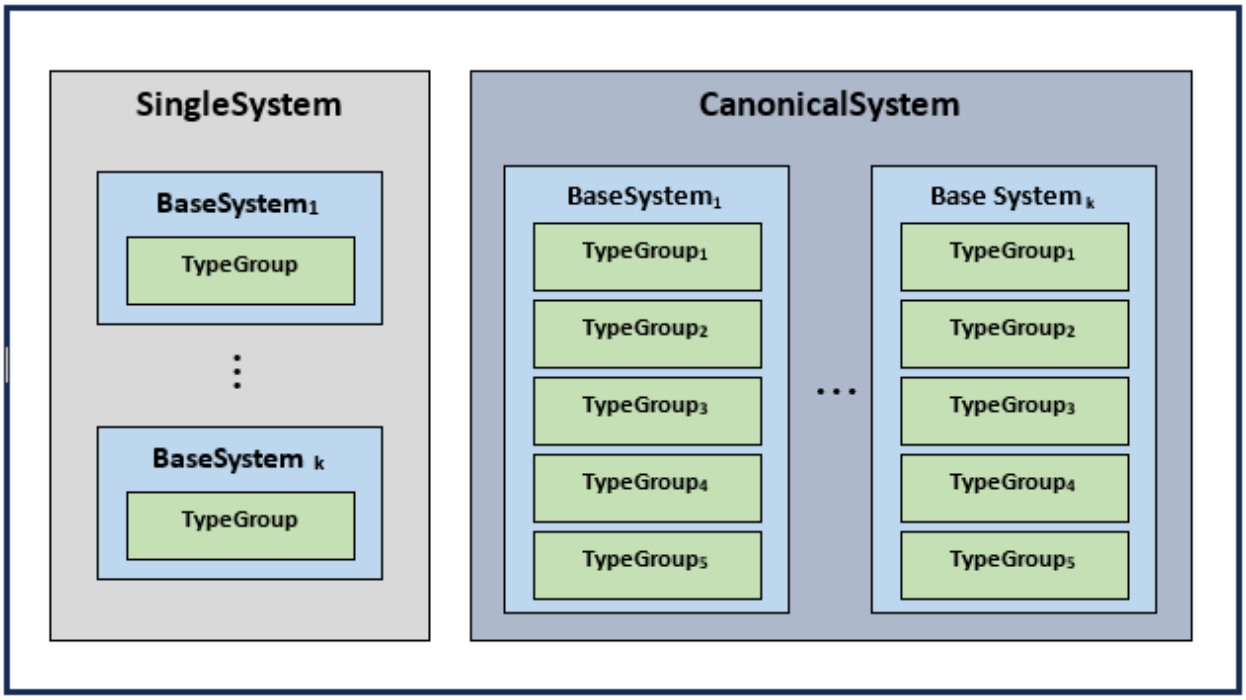

Figure 6.7: *Unit Foundation Classes.* A `SingleSystem` consists of one or more `BaseSystems` each contain a single TypeGroup. A CanonicalSystem consists of one or more `BaseSystems` each having five `TypeGroups`.

The `SingleSystem` and `CanonicalSystem` classes inherit from a common base class, `Conversion`. The `Conversion` class contains manipulation methods for adding and removing units (see §7, **Adding and Removing Units**) and performing unit conversions. This class is a child class of the `ConversionBase` class which contains accessor methods and the map (or dictionary) for holding the unit data. Figure 6.8 shows the class diagram for these classes.

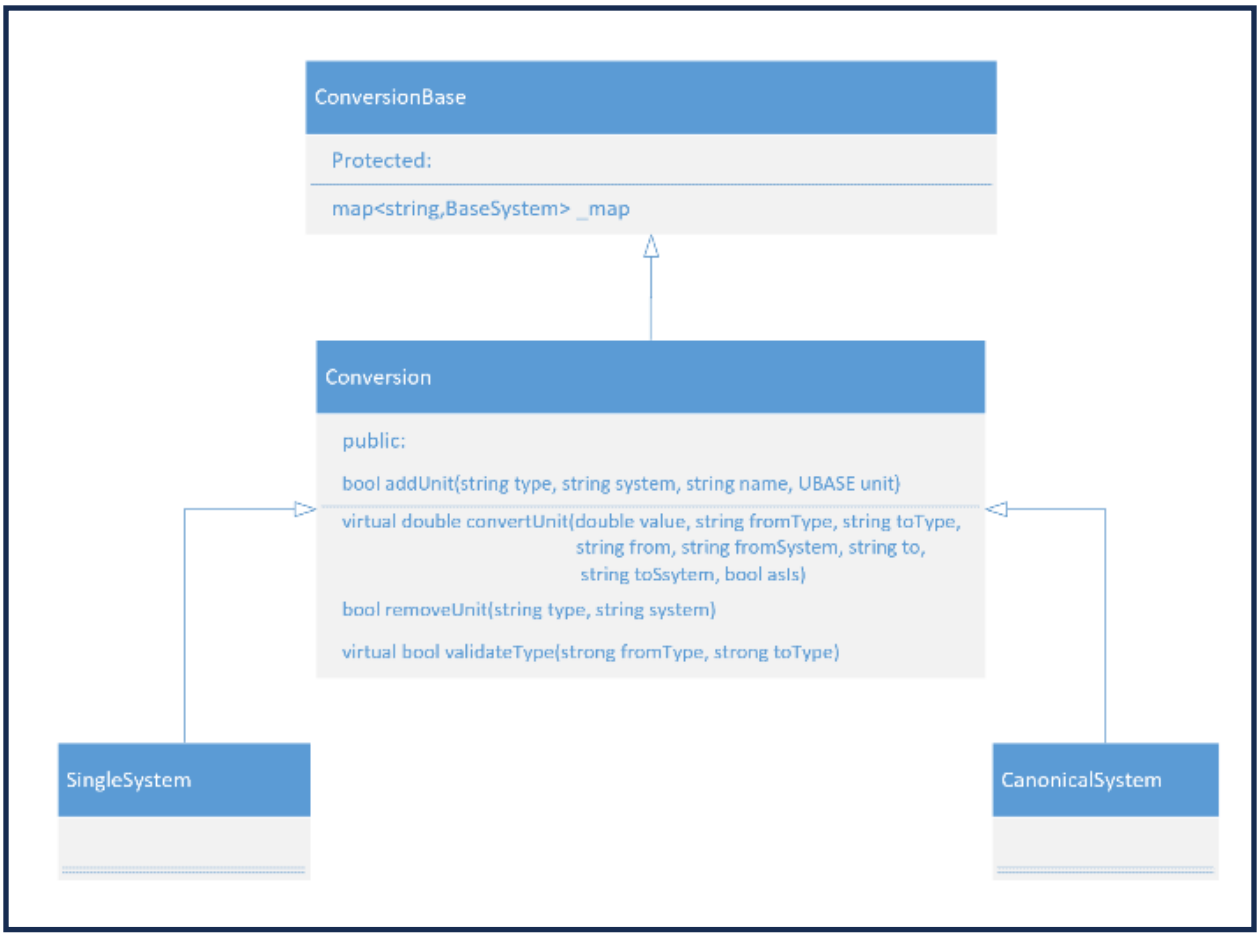

Figure 6.8: *Foundation Class Inheritance.* The `SingleSystem` and `CanonicalSystem` classes are the parent classes for the generated code. They are child classes for the Conversion class which contains modifier methods. His class inherits from the `ConversionBase` class which contains accessor methods.

The `SingleSystem` and `CanonicalSystem` classes are different ways of organizing the unit data. For unit classes that inherit from `SingleSystem` the organization is by *type*. Units in the containing map (or dictionary) are index by system and unit name. Unit classes that inherit from `CanonicalSystem` are organized by *system* then type and indexed by name. The reason for the two different organizations is that all unit systems contain canonical units but for ancient and most historical systems these are the only unit types. Modern systems have additional units for physical measurements like force, temperature, and electric and magnetic fields. As Table 9.3 shows there are about 90 unit-systems of which the overwhelming majority are ancient or historical. Grouping these systems together allows for better type separation for the more numerous (in type, but not number) modern units.

These two organizations introduce a transposition in the code when accessing the underlying units. This transposition is handled by creating virtual methods that switch type and system names in the two foundation classes. The accessor and modifiers in the base classes are mirrored in the interface classes which constitute the final layer in the class hierarchy. Here units are classified into three groups: units of measurement, physical and other constants, and system units.

The units of measurement are contained in the `SingleSystem` and `CanonicalSystem` classes but accessing them directly using these classes is awkward since the units are organized categorically not by type. This is remedied using the `Convert` and `Converter` classes. At the `SingleSystem` and

`CanonicalSystem` level unit conversion requires specifying the type, system, and unit names for the unit being converted and the unit it is being converted to. Specifying just the unit names would be ideal, but unfortunately this would require unique names for every unit across systems and types. To mitigate this problem the type and system names can be encapsulated to reduce the amount of information required to specify the units for the conversion.

The `Convert` class contains a `Conversion` class as a member. This is either a `SingleSystem` or `CanonicalSystem`. It has members for specifying default from and to units systems. The from-system is the system of the unit being converted and the to-system is the system to which the converted unit belongs. Default systems are specified at construction but these can be changed using modifier methods. The `Convert` class also contains accessor methods for obtaining unit types, systems, and names. Table 6.3 provides a summary of these methods.

Table 6.3: Convert Methods

Method	Description
`allSystemNames`	Returns a list of all system names in the `Conversion` contained in the converter.
`allUnitNames`	Returns a list of all unit names in the `Conversion` contained in the converter.
`check`	Compares the software version to the unit version and returns `true` if these are the same.
`fromSystem`	Returns the name of the from-system.
`fromSystem`	Sets the name of the from-system.
`name`	Returns the name of the converter. This is typically a unit type.
`toSystem`	Returns the name of the to-system.
`toSystem`	Sets the name of the to-system.
`typeNames`	Returns a list of all the unit types in the converter.
`valid`	Returns true if the converter contains a valid `Conversion`.

The `check` method provides a mechanism for verifying that the units in the Excel spreadsheet match the version of the software in which they are contained. This method returns `true` if the versions match. If `false` is returned then there is a mismatch between the two and the library should not be used.

The `Converter` class inherits from the `Convert` class and is the interface for converting units. A `Converter` class is created for each unit type. The type can only be changed to another type having the same dimensions. This change-of-type currently applies only to volume which is split into liquid and dry volume since many systems use identical units names when referring to these two versions of the volume measurement. For example, the US system of measurement has a liquid and a dry volume which are both called *gallon*. These two gallons have *different* values with the dry gallon being larger than the liquid gallon. However, both units have the same dimensions so conversions between them are possible and therefore allowed by the library. A check is performed when an attempt is made to change type. If the type change would require a change of dimension, then it is rejected.

Table 6.3 provides a summary of the `Converter` class methods. The most notable of these are the `convert` methods which performs the unit conversions. There are two versions of this method. The first uses the default from and to systems and requires that only the unit names be specified. The second

requires specification of both system and unit names. For example, to convert 3.5 kilometers to miles the following syntax is used:

```
mi  = length.convert(3.5, "kilometer", "meter")
```

where the default from-system is 'SI" and the default to-system is 'Imperial'. If the default from and to systems have been configured differently, then the second version of the method can be used:

```
mi = length.convert(3.5, "kilometer", "SI", "mile", "Imperial")
```

In this version the from and to system are specified along with the corresponding unit names. For additional examples see §4, **Quick Start**.

Table 6.4: Converter Methods

Method	Description
`addUnit`	Add a unit to the converter (see §7, **Adding and Removing Units**)
`convert`	Convert a unit using default from and to systems.
`convert`	Convert a unit specifying from and to systems.
`fromType`	Return name of from-type.
`fromType`	Set name of from-type.
`removeUnit`	Remove a unit from the converter (see §7, **Adding and Removing Units**)
`systemNames`	Return a list of unit system names in the converter.
`toType`	Return the name of converter to-type.
`toType`	Set name of converter to-type.
`type`	Return the converter unit type name.
`typeGroup`	Return a list of type groups in the converter.
`unit`	Return a unit from the converter.
`unitNames`	Return a list of all unit names in the converter.
`unitNames`	Return a list of all units names in a unit system contained in the converter.

The `Converter` class contains methods for returning lists of unit and system names and accessing both type groups and individual units. The `unit` method returns a `UBASE` object which can be queried for information such as system and type for an individual unit in the `Converter`. Units may also be added and removed from the `Converter` using the add and remove methods. For more information on adding and removing units, see §7, **Adding and Removing Units**.

A `Converter` object is created for each unit type. This is handled by the `UnitConverions` class. This class is a singleton. Its private constructor initializes a map (or dictionary) of `Converter` objects, one for each unit type. The static `Instance` method returns the single instance of the class and this instance is used to retrieve the desired `Converters`. Two additional methods are included: a `check` method to verify that unit and software versions match and a `names` method which returns the names of all the available `Converters`.

To use the Unit Conversion Library, first obtain the single instance of the `UnitConverions` class by calling its static `Instance` method, and then use the `converter` method to obtain the desired `Converter`(s). see §4, **Quick Start** for complete examples in the four supported languages.

Constants and System Units

Constants and system units are part of the library and follow a similar, but simpler class structure as used for units. The difference here is that constants and system units are immutable and cannot be converted. The `ConstantGroup` class plays a similar role for constants as the `TypeGroup` class plays for units. It stores a related group of constants in a map (or dictionary) indexed by the constant name. Constant values are sored as `UBASE` objects. `ConstantGroup` Includes methods for accessing constant names, system names, and type names as well as individual constant values. Table 6.5 provides a summary of each method.

Table 6.5: ConstantGroup Methods

Method	Description
`check`	Returns true if software and unit versions match.
`constant`	Returns a named constant as a `UBASE` object.
`constantNames`	Reruns a list of all constant names in the `ConstantGroup`.
`name`	Returns the name of the constant group.
`systemNames`	Reruns a list of all system names in the `ConstantGroup`.
`typeNames`	Reruns a list of all type names in the `ConstantGroup`.
`valid`	Returns true if the `ConstantGroup` contains one or more valid constants.
`value`	Returns the value of the named constant as a double precision number.
`version`	Returns the software version.

The `ConstantGroup` class also plays the role of the `Converter` class for units. A `ConstantGroup` is created for each class of constant (physical constants, physical measurements, physical definitions, WGS84 model) as well as the SI units and their derived units. The constant groups are placed in a map in the `Constants` class. The `Constants` class is a singleton with a private constructor that creates these instances of `ConstantGroup` for the various collections of constants. A static `Instance` method returns the single instance of this class. The `constant` method returns a named `ConstantGroup`. The `check` and `names` methods, respectively, verify software and unit versions and list the `ConstantGroup` names.

System units are similar to constants in that they are immutable and cannot be converted. However structurally they are canonical units since they form the base for those (and other) units. As such the generated class `SYSTEM_UNITS` inherits form the library class `SystemUnits` which is a child class of `ConversionBase`. The `SystemUnits` class is essentially a version of the `CanonicalSystem` class without the modifier methods. Table 6.6 provides a summary of its methods.

Table 6.6: SystemUnits Methods

Method	Description
`unitNames`	Return a list of all unit names in the unit system.
`UnitNames`	Return a list of all unit names in a specified unit system.
`typeNames`	Returns a list of all the unit types in the unit system.
`value`	Returns the value of a named system unit as a double precision number.

The `SYSTEM_UNITS` class is the only generated class that should be called directly. The static class `Instance` method is used to obtain and instance of this singleton class. The methods in the parent classes `SystemUnits` and `ConversionBase` are used to access names and unit data. Figure 6.9 show class diagrams for both constants and system units.

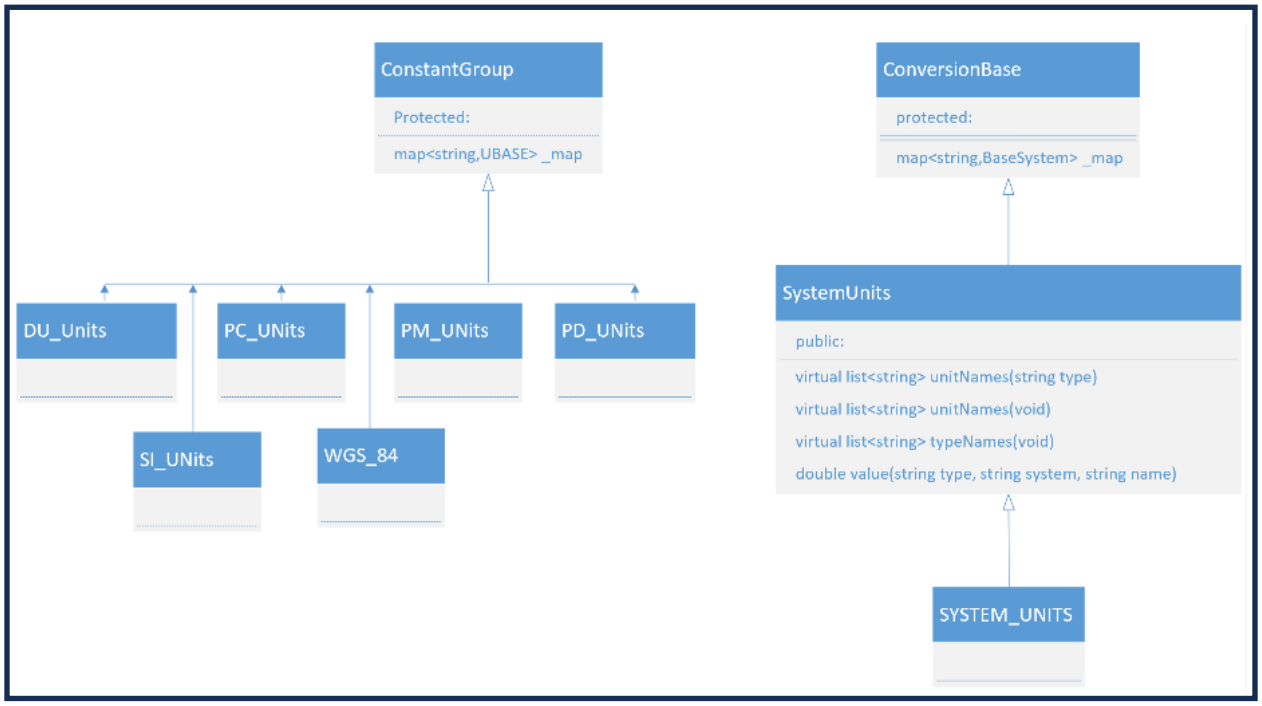

Figure 6.9: *Constant and System Units Classes.* The various constant classes in the library inherit from the `ConstantGoup` class which provides accessor functions for their constants. The generated class `SYSTEM_UNITS` inherits from `SystemUnits` and `ConversionBase` which provide accessor methods for these canonical base units.

Error Handling

The object-oriented method for handling errors is to throw and exception. All of the supported languages support exception and exception handling. This said, the throwing of exceptions by third party libraries and other such software leaves the user in the usually perplexing situation of trying to determine whose error resulted in the exception. Is it a coding error in the third-party software or is that software being used incorrectly?

Since there is little complex processing in the unit conversion library software, most errors are a result of trying to access something that is not in the library. Typically, this happens when a unit name, system, or type is misspelled or a variation of a unit name is used that is not recognized by the software. Although throwing an exception in such circumstances is certainly appropriate, it interrupts the larger flow of the program and forces the use of try-catch statements which can clutter the code, often unnecessarily.

The unit conversion library uses a more old-school approach. For methods that return information, such as a list or object, an empty list or invalid object is retuned. Most classes in the library have a `valid` method that can be used to determine if the object contains useful data. Lists can be checked to see if

they are empty. For the methods that return a double precision number such as the `convert` methods, `NaN` is returned if there is an error. The interface classes `Converter`, `ConstantGroup`, and `SystemUnits` all contain an `isError` method that takes a double and returns `true` if the value is `NaN`.

The user if left to decide whether to check (or not) return values from the library methods. Since the `convert` method returns a `double` it can be used inline if desired and the composite value checked for errors, if desired. For lists, the size or length of the list can be checked to see it is greater than zero. Library methods that return lists are largely informational since they give the names of units, unit systems, or unit types. For methods that return objects such as instances of `UBASE` or `TypeGoup`, the `valid` methods are available to check if the returned object is valid i.e. contains information. All classes except the container classes `Constants` and `UnitConversion` have a `valid` method.

Testing

The Unit Conversion Library comes with a `Test` folder which contains four sub-folders, one for each of the supported languages. These folders contain test code that can be compiled, linked, and executed to run unit and system tests. Four folders are of note: the `BaseTest` folder which contains the parent test class which all tests inherit from, the `MainTest` folder which contains the main program (runs all the tests), the `SystemTests` folder which contains the code for the system tests, and the `UnitTests` folder which contains the unit tests. The `TestOutput` folder holds the test results.

The unit test code tests all the public methods in all the library classes. Each method has at least one test to verify the correctness of its results. The system tests are for the interface classes (`Constants`, `SystemUnits`, and `UnitConversions`) and verify the basic functionality of their methods.

An additional folder named `CompTests` performs more basic testing and makes a comprehensive self-check by performing all possible unit conversions in the library. This test can take several minutes to half an hour to run depending on language and CPU.

There are three command line flags for running these tests. The first is `FULL` which will run both the unit and system tests, and the comprehensive tests. The second is `ALL` which will run all the tests (system, unit, and comprehensive), and the third is `COMP` which will just run the comprehensive tests. If no flag is specified only the units and system tests are run.

Tests that fail are marked with `***ERROR***` to indicate that a bug exists in the software. The Python script Check.py located in the top-level directory of the `UnitConversion` folder (library package) checks all the files in all four of the `TestOutput` folders and prints a summary to the screen. Each line of each test output files is read and checked to verify if `***ERROR***` is present or not. Lines that are so marked are printed in the summary. If no errors are found then the statement '`No errors found`' is printed. A list of the files examined for each language is also printed (see Figure 6.10 for an example output).

Testing of course, can never be too thorough or comprehensive and no guarantee is provided that the Unit Conversion Library is free of defects and bugs (§2, **License**). That said, the library has been tested to a reasonable level of confidence and should perform as expected. The user is free to perform more testing and to expand and modify the test code to achieve whatever level of testing confidence is desired.

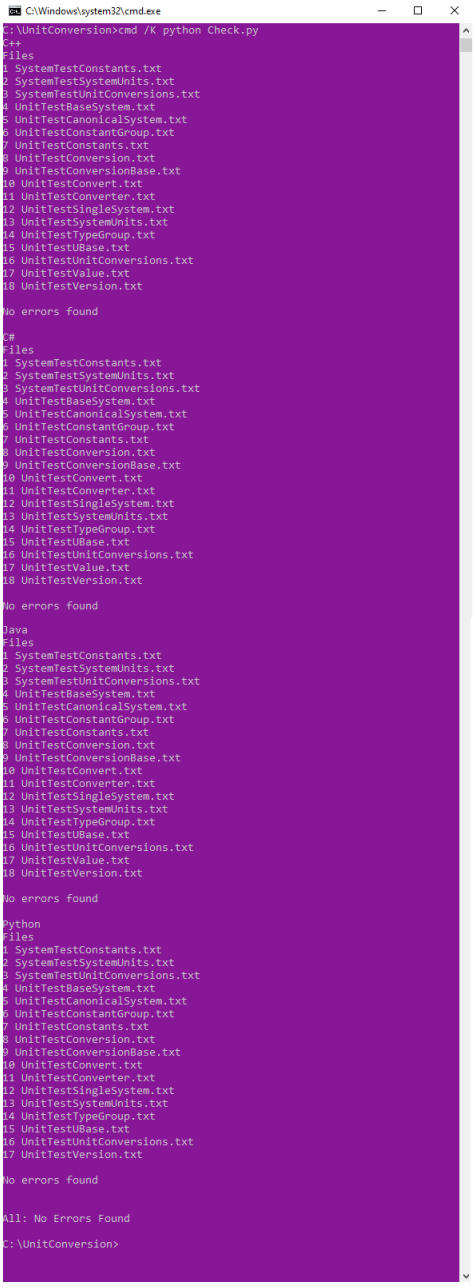

Figure 6.10: *21Checking Test Results.* The Check.py script reads the test output from the `TestOutput` folder for all four supported languages and checks for failed tests. A summary is printed to the command window.

Uniformity

There is a purposeful and intended uniformity in Unit Conversion Library code across the four supported languages. In any class, identical methods and members will be found in each language. The implementation of each method necessarily varies because of language syntactical differences, but in general there is an obvious and intentional correspondence in structure and style across the code. Anyone looking at the C++ implementation and comparing it to the Java or C# or Python implementation will see these similarities in style, structure and implementation.

This approach has the advantage of making implementation and maintenance of the library code base simpler and easier. It has the disadvantage of potentially not doing computations in the most efficient or language-like way. For example, C# uses properties instead of declaring a class member and providing accessor methods to retrieve and set its value. Since C++, Java, and Python do not have this feature, there are no properties in the C# code so that uniformity is maintained. This is of course, and arguable choice but the benefits of uniformity hopefully outweigh any small losses in efficiency or syntactical conformity.

7.0 Adding and Removing Units

There are two ways to add or remove units from the Unit Conversion Library. The first is statically and persistently by adding (or removing) a unit from the Excel spreadsheet and the second is dynamically and temporarily using the `addUnit` and `removeUnit` methods in the interface. Adding or removing units from the Excell spreadsheet obviously requires a working copy of Microsoft Excel to open the spreadsheet and add (or remove) the desired unit.

Using Excel

Adding or removing a unit requires finding the correct unit category and type. Units are grouped by type and placed into categories (worksheets) of similar unit types. Table 7.1 gives a mapping of unit type to worksheet.

Table 7.1: Unit Type, Dimensions, and Categories

Unit Type	Dimensions	Worksheet	Unit Type	Dimensions	Worksheet
Angular Units			**Energy**		
Angle	α	Angle	Energy	$M.L^2/T^2$	Energy
SolidAngle	Ω	Angle	Heat Flux	M/T^3	Energy
AngularVelocity	α/T	Angle	Thermal Conductance	$M.L^2/T^3.\Theta$	Energy
AngularAcceleration	α/T^2	Angle	Heat Transfer	$M/T^3.\Theta$	Energy
Computer Units			Thermal Conductivity	$M.L/T^3.\Theta$	Energy
Computer	b	Computer	Radiant Energy	M/T^2	Energy
ScreenResolution	pix	Computer	Entropy	$M.L^2/T^2.\Theta$	Energy
Voxel	vox	Computer	Molar Entropy	$M.L^2/T^2.\Theta.N$	Energy
Counts			Heat Insulation	$T^3.\Theta/M$	Energy
Count	O	Count	Energy Consumption	$L.T^2/M$	Energy
Density			Specific Heat Capacity	$L^2/T^2.\Theta$	Energy
LinearDensity	M/L	Density	Specific Heat	$L/T^3.\Theta$	Energy
SurfaceDensity	M/L^2	Density	**Frequency**		
Density	M/L^3	Density	Frequency	1/T	Frequency
Flow			**Luminosity**		
Flow	L^3/T	Flow	Luminous Intensity	J	Luminosity
Newtonian			Luminous Flux	$J.\Omega$	Luminosity
Force	$M.L/T^2$	Force	Luminance	J/L^2	Luminosity
Momentum	M.L/T	Force	Illuminance	$J.\Omega/L^2$	Luminosity
Action	$M.L^2/T$	Force	Radiant Intensity	$M.L^2/T^2.\Omega$	Luminosity
Mass Flow	M/T	Force	**Radiation**		
Quad Moment	L^4	Force	Particle Flux	$1/L^2.T.\Omega$	Radiation
Velocity	L/T	VelAccJrk	Specific Energy	L^2/T^2	Radiation
Acceleration	L/T^2	VelAccJrk	Radioactive Concentration	$1/L^3.T$	Radiation
Jerk	L/T^3	VelAccJrk	Exposure Rate	1/M	Radiation
Wavenumber			Specific Radioactivity	1/M.T	Radiation
Wavenumber	1/L	Wavenumber	Exposure	T.I/M	Radiation

Unit Type	Dimensions	Worksheet	Unit Type	Dimensions	Worksheet
Electricity and Magnetism			**Molarity**		
Current	I	EM	Molarity	N/L^3	Molarity
Charge	T.I	EM	Surface Molarity	N/L^2	Molarity
Magnetic Field Strength	I/L	EM	Molar Density	L^3/N	Molarity
Permeability	T/L	EM	Molar Absorption	L^2/N	Molarity
Magnetic Dipole Moment	$L^2.I$	EM	Radiochemical Yield	$T^2.N/M.L^2$	Molarity
Current Density	I/L^2	EM	Quantity	N	Molarity
Surface Charge Density	$T.I/L^2$	EM	Enzymatic Activity	N/T	Molarity
Charge Density	$T.I/L^3$	EM	Molar Charge	T.I/N	Molarity
Magnetic Reluctance	$T^2.I^2/M.L^2$	EM	**Specific Density**		
Electric Elastance	$M.L^2/T^4.I^2$	EM	Specific Length	L/M	SpecificDensity
Electric Conductance	$T^3.I^2/M.L^2$	EM	Specific Volume	L^3/M	SpecificDensity
Capacitance	$T^4.I^2/M.L^2$	EM	**Viscosity**		
Conductivity	$T^3.I^2/M.L^3$	EM	Kinematic Viscosity	L^2/T	Viscosity
Specific Acoustic Impedance	$M/L^2.T$	EM	Dynamic Viscosity	M/L.T	Viscosity
Magnetic Induction	$M.L^2/T^2.I$	EM	**Canonical Units**		
Inductance	$M.L^2/T^2.I^2$	EM	Length	L	Factors
Potential	$M.L^2/T^3.I$	EM	Area	L^2	Factors
Resistance	$M.L^2/T^3.I^2$	EM	Mass	M	Factors
Resistivity	$M.L^3/T^3.I^2$	EM	Liquid Volume	L^3	Factors
Acoustic Impedance	$M/L^4.T$	EM	Dry Volume	L^3	Factors
Electric Field Strength	$M.L/T^3.I$	EM	**Power and Pressure**		
Magnetic Flux Density	$M/T^2.I$	EM	Power	$M.L^2/T^3$	Power
Electric Field Gradient	$M/T^3.I$	EM	Pressure	$M/L.T^2$	Pressure
Magnetic Potential	$M.L/T^2.I$	EM	**Temperature and Time**		
Magnetic Permeability	$M.L/T^2.I^2$	EM	Temperature	Θ	Temperature
			Time	T	Time

Once the correct worksheet and location of the unit have been located in the Excel spreadsheet, the unit can be added or deleted. To delete the unit, just delete the corresponding row in the spreadsheet. To add a unit, add a new row and inset the unit information. Figure 6.3 shows an example of a typical worksheet in the workbook. For a new unit, Table 7.2 shows the required information that must be entered into the worksheet.

Unit names and abbreviations must be unique within a system and type. If there is more than one abbreviation for the unit, they must be separated by a comma in column B. The physical quantity

(column C) should match that of the units where the new unit is being added. The conversion factor (column E) should be calculated from its base value (column F). The base value should be calculated from cells in the `SI` and `Systems` worksheets. The value in column G, Unit, should match those of the units where the new unit is added. Finally, the system the unit belongs to should be placed in column H. The system name should be one of those in Table 9.3. If a new system is being added then its canonical units should be added the `System` worksheets.

Table 7.2: 12Worksheet Columns

Column	Title	Description
A	Unit Name	The unit name. Must be unique within a system and type.
B	Symbol	Abbreviation(s) for unit name. Must be unique within a system and type. Comma separated if more than one.
C	Physical Quantity	Unit's physical measurement e.g. length, force, energy, etc.
D	Dimensions	SI dimensions for the unit e.g. L, M.L/T2, M.L2/T2, etc.
E	Conversion Factor	Unit value in SI units.
F	Base	Base for calculating the unit value.
G	Unit	SI units e.g. m, kg.m/s2, kg.m2/s2, etc.
H	System	System unit belong to e.g. Imperial, US, UK, Roman, etc.

Calculating the base value from the `System`, `SI`, and the other tabs containing constants is somewhat arbitrary in the decision of what to include in the calculation. By way of example consider the Bohr radius which is given by the formula:

$$a_o = \frac{4\pi\varepsilon_o\hbar^2}{m_o e^2}$$

where:

ε_o is the permittivity of free space;
$\hbar$ is Planck's constant divided by 2π;
m_o is the electron rest mass;
e is the elemental electron charge.

The base for this unit is the unit itself and it is calculated as follows from values stored in the Physical Constants and Measurements worksheets:

```
'Physical Measurements'!E2*power('Physical Constants'!E2,2)/(PI()*'Physical Measurements'!E4*power('Physical Constants'!E4,2))
```

where:

'Physical Measurements'!E2 is the permittivity of free space;
'Physical Measurements'!E4 is the electron rest mass;
'Physical Constants'!E2 is Planck's constant;
'Physical Constants'!E4 is the elemental electron charge.

Once the unit has been added or removed from the Excel spreadsheet, the generated code containing the unit classes (which contain the unit data from the spreadsheet) needs to be regenerated. This is accomplished by clicking on the `Generate` button in the `Generate` worksheet (see Figure 6.2). The generated code is placed into the `Output` folder which is located at the same level as the Excel spreadsheet. This folder must be present before using the `Generate` button. The Python script `Distribute.py` can be run to place these files into their respective folders in the `UnitConversionLibrary` folder.

For C++ and C# users the code in the unit conversion library needs to be recompiled to generate new binaries. For Visual Studio users this can be done by opening the corresponding solution in the `Dev` folder and rebuilding. Other users will have to create makefile (not supplied) or import the code into whatever environment is being used and recompile there.

For Java users, the code will again need compiling into `class` files. Windows `BAT` files are supplied for Windows users others will have to create corresponding script files to compile the new code files. For Python user compiling is not required since new `pyc` files will be generated the next time a Python script using the library is run. However, for Windows users the `Build.bat` files (co-located with the source files) are provided to generate these files, if desired.

Of course, the user's project will need to be recompiled and relinked so use the newly generated library code. This process permanently adds or removes a unit from the library and the changes persist across executions of the program.

Using `addUnit` and `removeUnit`

Units can also be added and removed dynamically from the library during execution of the program by using the `Converter` class `addUnit` and `removeUnit` methods. These methods dynamically add or remove a unit from the library at runtime but these modifications do not persist beyond program execution. This feature is perhaps most useful for Python users working directly in the interpreter who want to add (or remove) a unit on-the-fly. For users of other languages, it simplifies the process of adding or removing a unit at the expense of persistency and efficiency. The following example illustrates usage.

Suppose it is desired to add the unit `sloth` to the library. This unit is a velocity and corresponds to moving at the pace of a furlong per fortnight. A furlong is 660 feet or 201.168 meters. A fortnight is 14 days or 1,209,600 seconds. A furlong per fortnight is therefore $1.663095238095238 \times 10^{-4}$ meters per second. Assuming that `velocity` is an instance of the `Converter` class of type velocity, then:

```
sloth =  new UBASE("Imperial", "sloth", 1.663095238095238e-4, "m/s","L/T",version)

velocity.addUnit("Imperial", "sloth", sloth)
```

adds the unit `sloth` to the `velocity` converter. This unit can now be used for subsequent unit conversions involving velocity. In the above, `Imperial` is the unit system and `version` is the software version which can be obtained from the `version` method of the singleton `Version` class.

If at some point in the program it is decided that the `sloth` unit is no longer required, it may be removed by calling the `removeUnit` method:

```
velocity.removeUnit("Imperial", "sloth")
```

The `removeUnit` method works for any unit in the library, not just with units added using the `addUnit` method. Removing units in this manner does not alter the unit conversion library binaries and as such, any added or removed units will still be in the library for use in other programs or at the start of a new execution of the program.

8.0 Unit Naming Conventions

A unit is uniquely defined by its type, system, and name. Type names are listed in Table 9.2 and system names are listed in Table 9.3. In both cases, all type and system names start with a capital letter and there are no spaces in these names. For system names corresponding to a country like Sri Lanka a hyphen is added to the name in place of a space i.e. Sri-Lankan. There are a few exceptions to the capitalization convention. The centimeter-gram-second system is abbreviated as cgs my common convention. In addition, some systems such as atomic unis are abbreviated as a.u. Type names involving two words such as Solid Angle are concatenated into a single word i.e. SolidAngle.

There are far too many unit names to list in this user's manual. The accompanying Excel spreadsheet can be examined for a complete list of unit names. Lowercase letters are used for unit names, again with some exceptions. These typically involve units whose names contain a single letter e.g. R-unit. Again, unit names do not contain spaces.

In the case where a unit is named after a person such as Watt which is named after James Watt (1736-1819), or whose name contains the name of a person or other proper noun, the unit name is *not* capitalized. Thus, watt is used instead of Watt to refer to units of power.

Some unit names require further qualification to avoid ambiguity. An example of this is the energy unit kilocalorie whose value is temperature dependent. In this case, a qualifier is added next to the name in parenthesis to disambiguate the unit. Thus kilocalorie(4°C), kilocalorie(15°C), and kilocalorie(20°C) are the names of three kilocalorie energy units in the 'Scientific' system. Capitalization may be used in qualifiers, as noted this this example.

Some unit names have abbreviations which can be used in place of the full unit name. In addition to the usual abbreviations such as '`m`' for `meter` and '`sec`' for `seconds`, most units such as '`british-thermal-unit(therm)-per-hour-per-square-foot`' with longer names have shorter (and potentially more useful) abbreviations such as '`Btu(IT)/hr-ft`'.

For unit names (and abbreviations) whose region of origin does not use the Latin or Roman alphabet, English equivalent translations or transcriptions are used. These tend not to be unique so the names of these units should be checked against those in the Excel spreadsheet to determine which variation is supported. This is especially true for languages such as Chinese, Japanese, Indian, another eastern or Asian languages which employ non-Latin alphabets or other mechanisms for writing.

Some SI prefixes use Greek letters as abbreviations. Since programing languages use plain Roman text, these letters are not available for use in such abbreviations. Instead, the closest roman letter in appearance is used. Thus, for example, 'u' is used for μ, to abbreviate the prefix 'micro'.

Every effort has been made to provide a consistency and uniformity in the naming of unit types, systems, and names. The `check` button under `Worksheet Integrity` on the `Generate` worksheet (see Figure 6.2) checks for unit name, system, type, and value formatting and provides a list of worksheet row numbers that do not conform to most of the criteria mentioned above. This said, with over 5,000 units, some anomalies (i.e. 'errors') are to be expected and hopefully corrected in future versions.

9.0 Appendix

Table 9.1: Software Environment

Language	Version
C++	Microsoft Visual Studio Professional 2019 Version 16.9.4, C++ 17 Standard.
C#	Microsoft Visual Studio Professional 2019 Version 16.9.4 Microsoft .NET Framework Version 4.8.09037
Java	1.8.0_281
Python	3.9.1

Table 9.2: Unit Types

Supported Unit Types			
Angle	Conductivity	SpecificHeatCapacity	Pressure
SolidAngle	SpecificAcousticImpedance	SpecificHeat	ParticleFlux
AngularVelocity	MagneticInduction	Flow	SpecificEnergy
AngularAcceleration	Inductance	Force	RadioactiveConcentration
Computer	Potential	Momentum	ExposureRate
ScreenResolution	Resistance	Action	SpecificRadioactivity
Voxel	Resistivity	MassFlow	Exposure
Count	AcousticImpedance	QuadMoment	SpecificLength
LinearDensity	ElectricFieldStrength	Frequency	SpecificVolume
SurfaceDensity	MagneticFluxDensity	LuminousIntensity	Temperature
Density	ElectricFieldGradient	LuminousFlux	Time
Current	MagneticPotential	Luminance	Velocity
Charge	MagneticPermeability	Illuminance	Acceleration
MagneticFieldStrength	Energy	RadiantIntensity	Jerk
Permeability	HeatFlux	Molarity	KinematicViscosity
MagneticDipoleMoment	ThermalConductance	SurfaceMolarity	DynamicViscosity
CurrentDensity	HeatTransfer	MolarDensity	Wavenumber
SurfaceChargeDensity	ThermalConductivity	MolarAbsorption	Length
ChargeDensity	RadiantEnergy	RadiochemicalYield	Area
MagneticReluctance	Entropy	Quantity	Mass
ElectricElastance	MolarEntropy	EnzymaticActivity	Liquid
ElectricConductance	HeatInsulation	MolarCharge	Dry
Capacitance	EnergyConsumption	Power	Volume

Table 9.3: Unit Systems

Supported Unit Systems			
Astronomical (@)	Cypriot	Korean	Saudi-Arabian
Atomic Units (a.u.)	Czech	Latvian	Scientific
Abyssinian	Danish	Libyan	Scottish
Afghan	Dutch	Luxemburg	SI
African	East Med	Malagasy	Singaporean
Algerian	Egyptian	Maltese	Somalian
Angle	Eritrean	Mauritius	South-African
Annamese	Estonian	Mexican	Spanish
Arabic	Ethiopian	MKpS	Sri-Lankan
Argentinian	Finnish	MKS	Sumerian
Asian	FPS	Moroccan	Swedish
Attic	French	MTS	Swiss
Austrian	German	Myanmar	Syrian
Balearic	Greek	Nepalese	Taiwanese
Belgian	Guatemalan	Nicaraguan	Tamil
Brazilian	Guinean	Norwegian	Tatar
Bulgarian	Haitian	Omani	Thai
Burmese	Hebrew	Pakistani	Time
Byzantine	Honduran	Paraguayan	Tanzanian
Cambodian	Hungarian	Persian	Tunisian
Canadian (CAN)	Icelandic	Peruvian	Turkish
Ceylon	IEUS	Philippine	Turkmenian
cgs	Imperial	Polish	UK
Chilean	Indian	Portuguese	US
Chinese	Indonesian	Puerto-Rican	Venezuelan
Colombian	International (INT)	Roman	WGS84
Computer	Irish	Romanian	Yugoslavian
Costa-Rican	Italian	Russian	
Cuban	Japanese	Salvadoran	

Table 9.4: Physical Constants

Constant	Value	Units
planck-constant	6.62607015E-34	$kg.m^2/s$
speed-of-light	2.99792458E+08	m/s
elementary-charge	1.602176634E-19	A.s
boltzman-constant	1.380649E-23	$kg.m^2/s^2.K$
avogadro-constant	6.02214076E+23	1/mol

Table 9.5: Physical Definitions

Definition	Value	Units	Definition	Value	Units
atmosphere(standard)	101325	$kg/m.s^2$	roentgen	0.000258	s.A/kg
bar	100000	$kg/m.s^2$	dozen	12	count
minute	60	s	millimeter	0.001	m
hour	3600	s	centimeter	0.01	m
day	86400	s	hertz	1	1/s
year(common)	31536000	s	dyne	0.00001	$kg/m.s^2$
year(Julian)	31557600	s	decimeter	0.1	m
liter	0.001	m^3	gram	0.001	kg
kg-water	0.001	m^3	carat(metric)	0.0002	kg
ream	500	count	angstrom	0.0000000001	m
ream(old)	480	count	nautical-mile	1852	m
standard-gravity	9.80665	m/s^2	ream(perfect)	512	count
carat	0.0002	kg	astronomical-unit	149597870700	m
BTU(IT)	1055.05585262	$kg.m^{2/}s^2$	earth-equatorial-radius-WGS-84	6378137	m
thermochemical-calorie	4.184	$kg.m^{2/}s^2$	room-temperature	293.15	K
international-nautical-mile	1852	m	curie	$3.7 \text{ X } 10^{10}$	1/s
international-steam-table-calorie	4.1868	$kg.m^2/s^2$			

Table 9.6: Physical Measurements

Measurement	Value	Units	Measurement	Value	Units
permittivity-of-free-space	8.8541878128E-12	$s^4.A^2/kg.m^3$	proton-mass	1.67262192369E-27	kg
permeability-of-free-space	1.2566370620E-06	$kg.m/s^2.A^2$	solar-mass	1.98847E+30	kg
electron-mass	9.1093837015E-31	kg	density-of-mercury(23^0C)	1.3537700000E+04	kg/m^3
bohr-radius	5.29177210903E-11	m	density-of-sea-water	1.025E+03	kg/m^3
density-of-water(4^0C)	9.9983950000E+02	kg/m^3	gas-constant-for-air	286	$m^2/s^2.K$
density-of-water(60^0F)	9.9901410000E+02	kg/m^3	ratio-of-specific-heat-for-air	1.4	O
density-of-mercury(0^0C)	1.3595080000E+04	kg/m^3	BTU(Mean)	1055.87	$kg.m^{2/}s^2$
density-of-mercury(60^0F)	1.3556790000E+04	kg/m^3	calorie(15°C)	4.1855	$kg.m^2/s^2$
electron-radius	2.8179403262E-15	m	calorie(20°C)	4.19002	$kg.m^2/s^2$
loschmidt's-number	2.6867774E+25	$1/m^3$	calorie(4°C)	4.20450	$kg.m^2/s^2$
roentgen	0.000258	A.s/kg	calorie(mean)	4.19002	$kg.m^2/s^2$
dalton	1.6605390666E-27	kg			

Table 9.7: System Units

System	Length (meters)		Area (square-meters)		Mass (kilograms)	
	Unit	Value	Unit	Value	Unit	Value
Astronomical	au	1.495978707000000E+11	square-parsec	9.521406136918410E+32	solar-mass	1.988470000000000E+30
Atomic	Bohr-radius	5.291772109060850E-11	atomic-cross-section	8.797355429314450E-21	electron-mass	9.109383701500000E-31
UK	foot	3.048000000000000E-01	square-foot	9.290304000000000E-02	pound	4.535923700000000E-01
US	foot	3.048000000000000E-01	square-foot	9.290304000000000E-02	pound	4.535923380000000E-01
Imperial	foot	3.048000000000000E-01	square-foot	9.290304000000000E-02	pound	4.535923380000000E-01
Canadian	foot	3.048000000000000E-01	square-foot	9.290304000000000E-02	pound	4.535923380000000E-01
cgs	centimeter	1.000000000000000E-02	square-centimeter	1.000000000000000E-04	gram	1.000000000000000E-03
FPS	foot	3.048000000000000E-01	square-foot	9.290304000000000E-02	pound	4.535923700000000E-01
MKpS or MKfS	meter	1.000000000000000E+00	square-meter	1.000000000000000E+00	hyl	9.806650000000000E+00
MTS	meter	1.000000000000000E+00	square-meter	1.000000000000000E+00	tonne	1.000000000000000E+03
SI	meter	1.000000000000000E+00	square-meter	1.000000000000000E+00	kilogram	1.000000000000000E+00

System	Liquid Volume (cubic-meters)		Dry Volume (cubic-meters)	
	Unit	Value	Unit	Value
Astronomical	cubic-parsec	2.937998946096350E+49	cubic-parsec	2.937998946096350E+49
Atomic	atomic-volume	6.207146679245500E-31	atomic-volume	6.207146679245500E-31
UK	gallon	4.546092000000000E-03	gallon	4.546092000000000E-03
US	gallon	3.785411784000000E-03	gallon	4.404883770860000E-03
Imperial	gallon	4.546092000000000E-03	gallon	4.546092000000000E-03
Canadian	gallon	4.546092000000000E-03	gallon	4.546092000000000E-03
cgs	liter	1.000000000000000E-03	cubi-centimeter	1.000000000000000E-06
FPS	gallon	3.785411784000000E-03	cubic-foot	2.831684659200000E-02
MKpS or MKfS	cubic-meter	1.000000000000000E+00	cubic-meter	1.000000000000000E+00
MTS	stere	1.000000000000000E+00	cubic-meter	1.000000000000000E+00
SI	cubic-meter	1.000000000000000E+00	square-meter	1.000000000000000E+00

System	Length (meters)		Area (square-meters)		Mass (kilograms)	
	Unit	Value	Unit	Value	Unit	Value
Abyssinian	pic	6.860000000000000E-01	square-pic	4.705960000000000E-01	rottolo	3.110000000000000E-01
Afghan	gaz-i-shah	1.065000000000000E+00	jareeb	2.000000000000000E+03	nakhud	1.900000000000000E+04
Algerian	pic (dzera-a-torky) pic (dzera-a-raby)	6.230000000000000E-01 4.670000000000000E-01	square-pic (dzera-a-torky) square-pic (dzera-a-raby)	3.881290000000000E-01 2.180890000000000E-01	ukkia	3.413000000000000E-02
Annamese	thuoc-moc thuoc-de-ruong thuoc-vai ngu	4.250000000000000E-01 4.700000000000000E-01 6.440000000000000E-01 2.000000000000000E+00	square-ngu thon	5.522500000000000E+00 2.400000000000000E+00	dong	3.775000000000000E-03
Arabic	foot	3.200000000000000E-01	feddan	5.898240000000000E+03	rotl	3.400000000000000E-01
Argentinan	vara	8.666000000000000E-01	square-vara	7.509955600000000E-01	libra	4.594000000000000E-01
Attic	pous	3.085600000000000E-01	square-pous	9.520927360000000E-02	talent	2.592000000000000E+01
Austrian	fuss	3.160800000000000E-01	joch	5.754618224640000E+03	pfund	5.600100000000000E-01

System	Liquid Volume (cubic-meters)		Dry Volume (cubic-meters)	
	Unit	Value	Unit	Value
Abyssinian	madega	4.400000000000000E-04	cubic-pic	3.228288560000000E-01
Afghan	cubic-gaz-i-shah	1.207949625000000E+00	cubic-gaz-i-shah	1.207949625000000E+00
Algerian	khoul	1.666666666666670E-02	caffiso	3.174700000000000E-01
	metalli	1.790000000000000E-02		
Annamese	hao dau thang(cochinchina)	2.826000000000000E-02 1.000000000000000E-03 2.766000000000000E-03	than	1.600000000000000E+00
Arabic	sa	4.212500000000000E-03	cafiz	3.264000000000000E-02
Argentinan	frasco	2.375000000000000E-03	fanega	1.371977000000000E-01
Attic	cotyle	2.700000000000000E-04	cotyle chenix	2.700000000000000E-04 1.080000000000000E-03
Austrian	mass	1.415100000000000E-03	metzel	6.148900000000000E-02

System	Length (meters)		Area (square-meters)		Mass (kilograms)	
	Unit	Value	Unit	Value	Unit	Value
Belearic	canna	1.564000000000000E+00	square-canna	2.446096000000000E+00	rottolo	4.080000000000000E-01
Belgian	perche	6.497000000000000E+00	arpent	4.221100900000000E+01	livre	4.895000000000000E-01
Brazilian	pe	3.333333333333330E-01	salamis(nominal) tarefa(nominal)	4.537500000000000E+03 3.500000000000000E+03	libra	4.590500000000000E-01
Bulgarian	arshin	7.112000000000000E-01	square-arshin	5.058054400000000E-01	oka	1.270058636000000E+00
Burmese	sandong	5.588000000000000E-01	square-sandong	3.122574400000000E-01	catty	5.440000000000000E-01
Byzantine	pous	3.085000000000000E-01	pous	9.517225000000000E-02	ogkia	2.725000000000000E-02
Cambodian	muoi	1.000000000000000E+00	square-muoi	1.000000000000000E+00	neal	6.000000000000000E-01
Ceylonese	covid(Ceylon) covid(Madras)	4.640000000000000E-01 4.720000000000000E-01	square-covid(Ceylon) square-covid(Madras)	2.152960000000000E-01 2.227840000000000E-01	seer	2.834950000000000E-01
Chilean	bara	8.360000000000000E-01	square-bara	6.988960000000000E-01	libbra	4.600930000000000E-01
Chinese	tchi chek	3.200000000000000E-01 3.714750000000000E-01	meou fang-chi mau5	6.144000000000000E+02 1.024000000000000E-01 7.614000000000000E+02	jin	5.968160000000000E-01

System	Liquid Volume (cubic-meters)		Dry Volume (cubic-meters)	
	Unit	Value	Unit	Value
Belearic	quartera	7.197000000000000E-02	quartin	2.714000000000000E-02
Belgian	pot	5.000000000000000E-04	pot	1.500000000000000E-03
Brazilian	alquiera alquiera(Bahia)	5.324000000000000E-03 3.524000000000000E-03	cubic-pe alquiera(salt)	3.703703703703700E-02 4.076000000000000E-03
Bulgarian	krina	2.000000000000000E-02	cubic-arshin	3.597288289280000E-01
Burmese	byee	5.050000000000000E-04	cubic-sandong	1.744894574720000E-01
Byzantine	xestes	5.492800000000000E-04	cubic-pous	2.936063912500000E-02
Cambodian	sesep	4.000000000000000E-02	cubic-muoi	1.000000000000000E+00
Ceylonese	cubic-covid(Ceylon) cubic-covid(Madras)	9.989734400000000E-02 1.051540480000000E-01	chundroon	2.649000000000000E-04
Chilean	almude	8.083000000000000E-03	cubic-bara	5.842770560000000E-01
Chinese	cheng cyut3	1.035468800000000E-03 1.031000000000000E-03	cubic-tchi quartin	3.276800000000000E-02 2.714000000000000E-02

System	Length (meters)		Area (square-meters)		Mass (kilograms)	
	Unit	Value	Unit	Value	Unit	Value
Colombian	vara	8.000000000000000E-01	square-vara	6.400000000000000E-01	libbra	5.000000000000000E-01
Costa Rican	vara	8.393000000000000E-01	manzana	7.044244900000000E+03	fanega	9.200000000000000E+01
Cuban	vara	8.479536000000000E-01	caballera	1.342020000000000E+05	libra	4.608951746418000E-01
Cypriot	pic	6.096000000000000E-01	scala	1.337803776000000E+03	oke	1.270058636000000E+00
Czechoslovakian	stopa(Bohemian) stopa(Praha) stopa(Moravian) stopa(Silesian)	2.960000000000000E-01 2.965000000000000E-01 2.840000000000000E-01 2.895000000000000E-01	merice	2.000000000000000E+03	quintal	5.000000000000000E+01
Danish	fod danskmil	3.138570000000000E-01 7.532500000000000E+03	square-ruthe tondelande	1.418489516865600E+01 5.516230000000000E+03	pund	5.000000000000000E-01
Dutch	voeten	2.830594000000000E-01	morgen	8.244346000000000E+03	pond(Amsterdam) pond(ordinary)	4.940903200000000E-01 4.921677200000000E-01

System	Liquid Volume (cubic-meters)		Dry Volume (cubic-meters)	
	Unit	Value	Unit	Value
Colombian	cubic-vara	5.120000000000000E-01	cubic-vara	5.120000000000000E-01
Costa Rican	botella(nominal)	6.500000000000000E-04	cubic-vara	5.912234744570000E-01
Cuban	bocoy	1.362700000000000E-01	fanega	5.634727319684110E-02
Cypriot	oke	1.278550000000000E-03	cubic-pic	2.265347727360000E-01
Czechoslovakian	merice	7.060000000000000E-02	cubic-stopa(Bohemian) cubic-stopa(Praha) cubic-stopa(Moravian) cubic-stopa(Silesian)	2.593433600000000E-02 2.606598212500000E-02 2.290630400000000E-02 2.426306737500000E-02
Danish	pott	9.661520492510560E-04	korntonde	1.391258950921520E-01
Dutch	mingelen	1.200000000000000E-03	schepel	2.726000000000000E-02

System	Length (meters)		Area (square-meters)		Mass (kilograms)	
	Unit	Value	Unit	Value	Unit	Value
Eastern Mediterranean	pik	7.112000000000000E-01	square-pik	5.058054400000000E-01		
Egyptian	derah(royal-cubit) diraa derah(cubit) palm(short)	5.235000000000000E-01 5.800000000000000E-01 4.495800000000000E-01 7.485714286000000E-02	pekeis feddan-masri	2.740522500000000E+01 4.200080000000000E+03	deben oke	1.365000000000000E-02 1.248000000000000E+00
Eritrean	cubi	3.200000000000000E-01	square-cubi	1.024000000000000E-01	rottolo	4.480000000000000E-01
Estonian	arshine	3.071120000000000E+01	lofstelle tonnland	1.855000000000000E+03 5.462700000000000E+03	pfund	4.600000000000000E-01
Ethiopian	kend	4.900000000000000E-01	square-kend	2.401000000000000E-01	farasula(ivory) farasula(coffee) farasula(rubber)	1.347840000000000E+01 1.684800000000000E+01 1.797120000000000E+01
Finnish	jalka	2.969000000000000E-01	kannunala	8.814961000000000E+01	naula	4.250797024000000E-01

System	Liquid Volume (cubic-meters)		Dry Volume (cubic-meters)	
	Unit	Value	Unit	Value
Eastern Mediterranean	cubic-pik	3.597288289280000E-01	cubic-pik	3.597288289280000E-01
Egyptian	keddah	2.062500000000000E-03	khar	3.400000000000000E-02
Eritrean	messe	1.500000000000000E-03	cubic-cubi	3.276800000000000E-02
Estonian	hulmit kulimet	1.148000000000000E-02 1.150161280000000E-02	cubic-elle	9.619307168929790E+02
Ethiopian	kuba	1.016000000000000E-03	madega	4.400000000000000E-04
Finnish	tunna tuoppi	1.634900000000000E-01 1.308580960450000E-03	tuoppi	1.327400000000000E-03

System	Length (meters)		Area (square-meters)		Mass (kilograms)	
	Unit	Value	Unit	Value	Unit	Value
French	toise	1.949036500200000E+00	pied-carre	1.055206466419960E-01	livre(de-Paris)	4.895058500000000E-01
	pied(metric)	3.333333333333330E-01	journal(Nantes)	2.671000000000000E+03	livre(Charlemange)	3.671280000000000E-01
	pied(de-Paris/du-Roi)	3.248394167000000E-01	ares	1.000000000000000E+02	livre(metric)	1.000000000000000E+00
	bourgeois	3.175000000000000E-03	exots(Agen)	1.687500000000000E+01		
	canne(Provence)	1.968500000000000E+00	dextres(Montpellier)	1.920000000000000E+01		
	point-didot	3.759715471075000E-04	corterade(Montpellier)	2.877000000000000E+03		
	pied(commum)	2.236067977500000E-01	journal(Bretagne)	4.863000000000000E+03		
	pied(d'ordonance)	3.248400000000000E-01	becheree(Lyonnais)	1.367000000000000E+03		
	toise(du-Chatelet)	1.949090000000000E+00	journal(Lyonnais)	4.000000000000000E+03		
German	fuss	3.138570000000000E-01	klafter	1.440020334575000E+02	pfund	4.677110000000000E-01
					zollpfund	5.144821000000000E-01

System	Liquid Volume (cubic-meters)		Dry Volume (cubic-meters)	
	Unit	Value	Unit	Value
French	pinte	9.521462584750000E-04	setier	1.516800000000000E-01
	chopine(Provence-wine)	2.299250000000000E-04	picotins	3.125000000000000E-03
German	quart	1.145069095408660E-03	metzen	3.705960000000000E-02
			metze	3.435890000000000E-03

System	Length (meters)		Area (square-meters)		Mass (kilograms)	
	Unit	Value	Unit	Value	Unit	Value
Greek	piki(short) piki(long) amma	6.480000000000000E-01 6.690000000000000E-01 2.100000000000000E+01	stremma	1.000000000000000E+03	oka	1.280000000000000E+00
Guatemalan	vara	8.359000000000000E-01	manzana	6.987288100000000E+03	fanega	9.200000000000000E+01
Guinean	pic	5.780000000000000E-01	square-pic	3.340840000000000E-01	benda	6.420000000000000E-02
Haitian	toise	1.948800000000000E+00	carreau	1.292300000000000E+03	gwo-mamit	1.700000000000000E+00
Hebrew	cubit cubit(acared)	5.550000000000000E-01 6.400000000000000E-01	geris	3.141592653589790E-04	mina(sacred) mina(Talmudic)	8.500000000000000E-01 3.542000000000000E-01
Honduran	vara	8.128000000000000E-01	manzana	6.606438400000000E+04	fanega	9.200000000000000E+01
Hungarian	faust	1.053600000000000E-01	square-meile	6.978262126496050E+07	oka	1.329479142678000E+00
Icelandic	fet	3.138570000000000E-01	ferfaomur	3.546223792164000E+00	pund	5.000000000000000E-01

System	Liquid Volume (cubic-meters)		Dry Volume (cubic-meters)	
	Unit	Value	Unit	Value
Greek	oka(average) baril	1.336500000000000E-03 7.423600000000000E-02	sexte	5.400000000000000E-04
Guatemalan	botella(nominal)	6.500000000000000E-04	cubic-vera	5.840674122790000E-01
Guinean	cubic-pic	1.931005520000000E-01	cubic-pic	1.931005520000000E-01
Haitian	baril	1.000000000000000E-01	cubic-toise	7.401194422272000E+00
Hebrew	bath(old) bath(new)	2.937600000000000E-02 2.142000000000000E-02	ephah(old) ephah(new)	2.937600000000000E-02 2.142000000000000E-02
Honduran	botella(nominal)	6.500000000000000E-04	cubic-vara	5.369713131520000E-01
Hungarian	eimer	5.430000000000000E-02	cubic-faust	1.169572870656000E-03
Icelandic	cubic-fet	3.091686557603380E-02	pottar	9.661520492510560E-04

System	Length (meters)		Area (square-meters)		Mass (kilograms)	
	Unit	Value	Unit	Value	Unit	Value
Indian	hasta guz(Bombay) guz(Calcutta)	4.570000000000000E-01 6.858000000000000E-01 9.150000000000000E-01	cawnie square-guz(Bombay) square-guz(Calcutta)	5.400000000000000E+02 4.703216400000000E-01 8.372250000000000E-01	pala seer drona	4.700000000000000E-02 9.330400000000000E-01 1.320000000000000E+01
Indonesian	depa	1.700000000000000E+00	bahoe	7.096500000000000E+03	picul	6.176130250000000E+01
Irish	troighid	2.500000000000000E-01	achar	4.046856422400000E+03	penginn	4.000000000000000E-04
Italian	piedi-liprando	5.137700000000000E-01	giornata	3.800000000000000E+03	libbra	3.070000000000000E-01
Japanese	shaku	3.030303030303030E-01	tsubo	3.305785123966940E+00	kwan	3.750000000000000E+00
Korean	cheok	3.030303030303030E-01	pyeong	3.305785123966940E+00	gwan	3.750000000000000E+00
Latvian	elle	5.370000000000000E-01	kapp	1.486400000000000E+02	pfund	4.190000000000000E-01
Libyan	pic	6.800000000000000E-01	square-pic	4.624000000000000E-01	rottolo	5.128000000000000E-01
Malagasy	rahf	1.180084000000000E+00	square-rahf	1.392598247056000E+00	nanki	6.479891000000000E-04
Maltese	canna	2.088000000000000E+00	qasba-kwadra	4.359744000000000E+00	rottolo	7.937900000000000E-01
Mexican	vara	8.380000000000000E-01	fanega	3.566275929600000E+04	libbra	4.602463400000000E-01

System	Liquid Volume (cubic-meters)		Dry Volume (cubic-meters)	
	Unit	Value	Unit	Value
Indian	parah	1.101000000000000E-01	drona seer	1.320000000000000E-02 1.000000000000000E-03
Indonesian	kan	1.575100000000000E-03	cubic-depa	4.913000000000000E+00
Irish	meisrin	6.000000000000000E-01	cubic-troighid	1.562500000000000E-02
Italian	barile(Florence-oil) barile(Florence-wine)	3.343000000000000E-02 4.560000000000000E-02	cubic-piedi-liprando	1.356145303196330E-01
Japanese	sho	1.803906836964690E-03	cubic-shaku	2.782647410746580E-02
Korean	doe	1.803906836964690E-03	cubic-cheok	2.782647410746580E-02
Latvian	stoof	1.275200000000000E-03	kulmet	1.091062080000000E-02
Libyan	barile	6.249750000000000E-02	orba	7.692000000000000E-03
Malagasy	cubic-rahf	1.643382909778830E+00	bambou	2.001579185478780E-03
Maltese	salma	2.909440000000000E+02	tomna	1.818436800000000E-02
Mexican	cuartillo	4.562640000000000E-04	cuartillo	1.891800000000000E-03

System	Length (meters)		Area (square-meters)		Mass (kilograms)	
	Unit	Value	Unit	Value	Unit	Value
Mozambican	aldan	1.600000000000000E+00	square-aldan	2.560000000000000E+00	bahar	1.090000000000000E+02
Moroccan	pic	6.100000000000000E-01	square-pic	3.721000000000000E-01	rotal	5.075000000000000E-01
Myanmar	taung	4.572000000000000E-01	square-taung	2.090318400000000E-01	aseittha	4.082330000000000E-01
Nepalese	angul	1.905000000000000E-02	dam	1.987254090000000E+00	tola	1.166000000000000E-02
Nicaraguan	vara	8.128000000000000E-01	manzana	6.988921041000000E+03	fanega	9.200000000000000E+01
Norwegian	fod	3.137000000000000E-01	kvadrat-rode	9.840769000000000E+00	skaalpund	4.981000000000000E-01
Pakistani	karam	1.676400000000000E+00	square-karam	2.810316960000000E+00	tola	1.166375000000000E-02
Omani	muscat	9.939020000000000E-01	square-muscat	9.878411856040000E-01	maund	3.968933238000000E+00
Paraguayan	vara vara(old)	8.660000000000000E-01 8.385600000000000E-01	lifio lifio(old)	7.499560000000000E+01 4.883605057152000E+03	libbra libbra(old)	4.590000000000000E-01 4.600800000000000E-01
Persian	zereth farsakh	3.200000000000000E-01 5.486400000000000E+03	gar	1.474560000000000E+01	talent rottel	3.260000000000000E+01 4.600000000000000E-01
Peruvian	vara pie	8.359800000000000E-01 2.786600000000000E-01	fanegada topo	3.144881521800000E+03 2.705995833868800E+03	libbra	4.600900000000000E-01

System	Liquid Volume (cubic-meters)		Dry Volume (cubic-meters)	
	Unit	Value	Unit	Value
Mozambican	cubic-aldan	4.096000000000000E+00	cubic-aldan	4.096000000000000E+00
Moroccan	mud	1.400000000000000E-02	cubic-pic	2.269810000000000E-01
Myanmar	hkwet	1.278590000000000E-03	cubic-taung	9.556935724800000E-02
Nepalese	pathi	4.545960000000000E-03	cubic-angul	6.913292625000000E-06
Nicaraguan	botella	6.500000000000000E-04	cubic-vara	5.369713131520000E-01
Norwegian	pot	9.651000000000000E-04	korntonde	1.389744000000000E-01
Omani	ferren	3.000090300000000E-02	cubic-muscat	9.818173300541870E-01
Pakistani	cubic-karam	4.711215351744000E+00	cubic-karam	4.711215351744000E+00
Paraguayan	fanega	2.880000000000000E-01	cubic-vara	6.494618960000000E-01
Persian	sextario chenica	3.300000000000000E-04 1.320000000000000E-03	amphora	3.260000000000000E-02
Peruvian	cubic-vara	5.842351232431920E-01	cubic-vara	5.842351232431920E-01

System	Length (meters)		Area (square-meters)		Mass (kilograms)	
	Unit	Value	Unit	Value	Unit	Value
Philippine	talampakan	3.048000000000000E-01	balita	2.795000000000000E+03	catty	6.000000000000000E-01
Polish	stopa	2.880000000000000E-01	square-stopa	8.294400000000000E-02	funt	4.055040000000000E-01
Portuguese	pe	3.285000000000000E-01	square-vera	1.199025000000000E+00	libra	4.590000000000000E-01
Roman	pes(common)	2.944000000000000E-01	quadratus(common)	8.667136000000000E-02	uncia	2.725000000000000E-02
Romanian	halibiu palma	7.010000000000000E-01 2.500000000000000E-01	feredela	1.250000000000000E+00	cantar font	5.600000000000000E+01 5.000000000000000E-01
Russian	foute	3.048000000000000E-01	square-foute	9.290304000000000E-02	funt	4.095171792456690E-01
Salvadoran	vara	8.128000000000000E-01	square-vara	6.606438400000000E-01	fanega	9.200000000000000E+01
Saudi Arabian	farsakh	4.830000000000000E+03	square-covid	2.323240000000000E-01	maund	1.350000000000000E+00
Scottish	foot(traditional) foot	3.064460845920000E-01 3.064500000000000E-01	square-foot	9.390920276176720E-02	pound	6.168856232000000E-01
Somalian	top	3.920000000000000E+00	darat	8.000000000000000E+03	rottolo	4.480000000000000E-01

System	Liquid Volume (cubic-meters)		Dry Volume (cubic-meters)	
	Unit	Value	Unit	Value
Philippine	kaban	9.990000000000000E-02	cubic-talampakan	2.831684659200000E-02
Polish	kwarta	1.000000000000000E-03	cubic-stopa	2.388787200000000E-02
Portuguese	almude	1.650000000000000E-02	fanga	5.400000000000000E-02
Roman	sextarius	5.492800000000000E-04	modius	8.788480000000000E-03
Romanian	viacka litra	1.415000000000000E-02 2.500000000000000E-04	dimerla	2.460000000000000E-02
Russian	vedro	1.229941000000000E-02	garnetz	3.279842666666670E-03
Salvadoran	botella	6.500000000000000E-04	cubic-vara	5.369713131520000E-01
Saudi Arabian	nusfiah	9.500000000000000E-04	teman	8.500000000000000E-02
Scottish	gallon jug	1.355590372684800E-02 1.694918130000000E-03	lippy(oat-barley-malt) lippy(wheat-peas-beans-rice-salt)	3.283066337080000E-03 2.250484660312000E-03
Somalian	chela	1.359000000000000E-03	cubic-top	6.023628800000000E+01

System	Length (meters)		Area (square-meters)		Mass (kilograms)	
	Unit	Value	Unit	Value	Unit	Value
Spanish	vara	8.359050000000000E-01	square-vara	6.987371690250000E-01	libra	4.600930000000000E-01
Sri Lanka	paramaanuwa	3.306670000000000E-08	laaha	4.598700000000000E+00	gunja	1.200000000000000E-04
South Afican	cape-foot rijnlandse-voet	3.148580000000000E-01 3.139440000000000E-01	morgen	8.565320000000000E+03	pond	4.940903200000000E-01
Sumerian	kus	5.186000000000000E-01	gin	2.689459600000000E-01	gin	8.400000000000000E-03
Swedish	fot	2.969000000000000E-01	kvadratfot	8.814961000000000E-02	skalpund	4.250797024000000E-01
Swiss	fuss	3.000000000000000E-01	arpent	3.600000000000000E+01	livre	5.000000000000000E-01
Syrian	pic	5.820000000000000E-01	square-pic	3.387240000000000E-01	rottolo	1.785000000000000E+00
Taiwanese	chhioh	3.030303030303030E-01	pe	3.305785123966940E+00	nu	3.750000000000000E-02
Tamil	yaar	9.333123317760000E-01	cent	4.046856422400000E+01	palam	4.800000000000000E-02
Tanzanian	ohra	5.709920000000000E-01	square-ohra	3.260318640640000E-01	mane	9.104052458270000E-01
Tatar	sajin	2.133600000000000E+00	quadrat sajin	4.552248960000000E+00	qadaq	4.095000000000000E-01
Thai	wah	2.000000000000000E+00	square-wah	4.000000000000000E+00	tchang	1.200000000000000E+00
Tunisian	pic(Tunisian)	5.625000000000000E-01	square-pic	3.164062500000000E-01	uckir	3.149500000000000E-02
Turkish	pic	7.553972464870000E-01	square-pic	5.706250000001410E-01	oka	1.283000000000000E+00

System	Liquid Volume (cubic-meters)		Dry Volume (cubic-meters)	
	Unit	Value	Unit	Value
Spanish	arroba	1.564316200000000E-02	fanega	5.550100000000000E-02
Sri Lanka	seer	1.135623535200000E-03	seer	7.062500000000000E-04
South African	kanne firkin	1.329000000000000E-03 4.091482800000000E-02	schepel gantang	2.727500000000000E-02 9.200000000000000E-03
Sumerian	sila	1.000000000000000E-03	cubic-kus	1.394753748560000E-01
Swedish	kanna	2.617161920900000E-03	kanna	2.617161920900000E-03
Swiss	pot	1.500000000000000E-03	emine	1.500000000000000E-02
Syrian	rotl	3.200000000000000E-03	cubic-pic	1.971373680000000E-01
Taiwanese	liter	1.000000000000000E-03	cubic-chhioh	2.782647410746580E-02
Tamil	padi	1.344000000000000E-03	padi	1.344000000000000E-03
Tanzanian	cubic-ohra	1.861615860000000E-01	djzela	2.574214075690580E-01
Tatar	garnets	3.279733333333330E+00	cubic-sajin	9.712678381056000E+00
Thai	tanan(nomial)	1.000000000000000E-03	cubic-wah	8.000000000000000E+00
Tunisian	cafisso	4.960000000000000E-01	cubic-pic	1.779785156250000E-01
Turkish	fortin	4.000000000000000E-01	cubic-pic	4.310485537767510E-01

System	Length (meters)		Area (square-meters)		Mass (kilograms)	
	Unit	Value	Unit	Value	Unit	Value
Turkmenian	hasch/altschin	7.112000000000000E-01	square-hasch	5.058054400000000E-01	batman	1.265000000000000E+02
Venezuelan	vara	8.000000000000000E-01	square-vara	6.400000000000000E-01	libbra	5.000000000000000E-01
Welch	palf	3.048000000000000E-01	erw(Venedotian) erw(Dimetian)	1.426990694400000E+03 8.561944166400000E+02	pwys	4.535923380000000E-01
Yugoslavian	stopa	3.160000000000000E-01	square-stopa	9.985600000000000E-02	oka	1.280000000000000E+00

System	Liquid Volume (cubic-meters)		Dry Volume (cubic-meters)	
	Unit	Value	Unit	Value
Turkmenian	cubic-hasch	3.597288289280000E-01	cubic-hasch	3.597288289280000E-01
Venezuelan	arroba	1.613700000000000E-02	cubic-vara	5.120000000000000E-01
Welch	hestawr	7.047814033376000E-02	cubic-palf	2.831684659200000E-02
Yugoslavian	cubic-stopa	3.155449600000000E-02	cubic-stopa	3.155449600000000E-02

10.0 Acronyms

cgs – centimeter-gram-second

CPU – Central Processing Unit

DLL – Dynamic Link Library

GCC - GNU Compiler Collection

GNU - Gnu's Not Unix

IEUS – International Electrical Units

MKpS - Mètre–Kilogramme-poids–Seconde

MKS – Meter-Kilogram-Second

MTS – Meter-Tonne-Second

NaN – Not a Number

SI – Systéme International (International System)

UK – United Kingdom

US – United States

VBA – Visual Basic Applications

11.0 References

[1] Cardarelli, F., *Scientific Unit Conversion*, Second Edition, A Practical Guide to Metrication, Springer-Verlag, London, 1999, ISBN 1-85233-043-0.

[2] Cardarelli, F., *Encyclopedia of Scientific Units, Weights, and Measures*, Their SI Equivalences and Origins, Springer-Verlag, London, 2003, ISBN 978-1-4471-1122-1.

[3]Jerrard, H.G., McNeill, D.B., Dictionary of Scientific Units, Including Dimensionless Numbers and Scales, Sixth Edition, Springer-Science+ Business Media,B.V., 1992, ISBN 978-0-412-46720-2.

[4] Dresner, S., *Units of Measurement*, An Encyclopedic Dictionary of Units Both scientific and Popular and the Quantities They Measure, Harvey Miller and Medcalf Ltd, Aylesbury, England, 1971, ISBN 0856020028.

[5] Gupta, S.V., Units of Measurement, Past Present and Future International System of Units, Springer, 2009, ISBN 3642007376.

[6] Chandler, B, *User:Imaginatorium/Cardarelli*, Wikipedia, December 22, 202, https://en.wikipedia.org/wiki/User:Imaginatorium/Cardarelli.

[7] Wikipedia, *French Units of Measurement*, Wikipedia, July 20, 2023, https://en.wikipedia.org/wiki/French_units_of_measurement#cite_note-2

[8] Jespersen, J., Fitz-Randolph, J., *From Sundials to Atomic Clocks, Understanding Time and Frequency*, 2nd Edition, Dover Publications, 2011, ISBN 0486409139.

[9] Wikipedia, *Cubit*, Wikipedia, December 1, 2023, https://en.wikipedia.org/wiki/Cubit.

[10] Wikipedia, *Metre*, Wikipedia, January 1, 2024, https://en.wikipedia.org/wiki/Metre.

[11] Wikipedia, *System of units of measurement*, Wikipedia, December 2, 2023, https://en.wikipedia.org/wiki/System_of_units_of_measurement.