Units Library Documentation

Philip Top

May 16, 2024

BASICS

The Units library provides a means of working with units of measurement at runtime, including conversion to and from strings. It provides a small number of types for working with units and measurements and operations necessary for user input and output with units.

This software was developed for use in LLNL/GridDyn, and HELICS and is currently a work in progress (though getting close). Namespaces, function names, and code organization is subject to change though is fairly stable at this point, input is welcome.

The *Introduction* is a discussion about the why? and How the library came together and a generally what it does and how it was tested. The *Installation and Linking* guide is a discussion about linking and using the library, and the *User Guide* is the how-to about how to use the software library. For the details see Details and to try out some of the string conversions check out *Units on the Web* Finally *Application Notes* contains some discussions on particular applications and usages.

CHAPTER

ONE

INTRODUCTION

1.1 Why?

So why have another units library? This was something we poked at for a while before writing the library. There are number of other well designed C++ libraries but none of them met our needs. Some needs are pretty general, and others specific to power systems and electrical engineering.

1.1.1 Design Requirements

- have a units type that can be used in virtual function calls
- Handle unit conversions
- handle Per Unit operations and unit conversions
- handle complex units easily like *$/puMw/hr*
- Operate on strings with conversion to and from strings

Previously the library we were using met these requirements but only for a very limited set of units. This library functioned but was nearing the limits of maintainability and operation as new units were needed and other conversions were required which required adding direct conversions between the classes of units and the code was getting to be a mess. So, looking around, many of the existing unit libraries in C++ represent individual units as an individual type. This works wonderfully if you can know all the types you want to use ahead of time. In our case, many of the conversions depended on configuration or input files so the units being converted were not known at compile time. A few were but in general they were not. That led to an issue of how to do you pass that unit to a function that is meant to hide the internal unit in use, so having a type per unit did not seem functional from a coding or structural perspective.

Many of the dimensional analysis libraries, actually all as far as I can tell in C++, do not support string conversions. There are a few examples in Java or Python, but our code is written in C++ so that didn't seem workable either. Which led to the conclusion that this library is needed and some additional design considerations.

- A small single compact type to represent all units (no bigger than a double) if we wanted to use it in a number of contexts.
- Another measurement type that was interoperable with doubles and numbers. Many numerical calculations came from a *real* array and numeric solver libraries so there is no opportunity to modify those types. Which means, we need double operations with measurements if we want to use them.
- *constexpr* as much as possible since many units are known at compile time and we need to be able to generate complex units from other simpler units.
- The library should be compatible with a broad range of compilers including some older ones back to GCC 4.7.
- a fairly expansive list of predefined units to simplify operation

Speaking with others, a few items and contexts came up such as recipes, trade documents, other software package unit representations, standardized string representation. Sometimes a lot of precision is needed, other times this is not the case.

And it would be nice to be able to deal with uncertainties in measurements, commodities, and containers.

How It Works

Given the design requirements the choice was how to make a class that could represent physical units. The desire for it to be a compact class drove the decision to somewhat limit what could be represented to physically realizable units.

Unit Representation

The unit class consists of a multiplier and a representation of base units. The seven SI units + radians + currency units + count units. In addition a unit has 4 flags, per-unit for per unit or ratio units. One flag[i_flag] that is a representation of imaginary units, one flags for a variety of purposes and to differentiate otherwise similar units[e_flag]. And a flag to indicate an equation unit. Due to the requirement that the base units fit into a 4 byte type the represented powers of the units are limited. The list below shows the bit representation range and observed range of use in equations and observed usage

- meter:[-8,+7] :normal range [-4,+4], intermediate ops [-6,+6]
- kilogram:[-4,+3] :normal range [-1,+1], intermediate ops [-2,+2]
- second:[-8,+7] :normal range [-4,+4], intermediate ops [-6,+6]
- ampere:[-4,+3] :normal range [-2,+2]
- kelvin:[-4,+3] :normal range [-4,+1]
- mole:[-2,+1] :normal range [-1,+1]
- candela:[-2,+1] :normal range [-1,+1]
- currency:[-2,+1] :normal range [-1,+1]
- count:[-2,+1] :normal range [-1,+1]
- radians:[-4,+3] :normal range [-2,+2]

These ranges were chosen to represent nearly all physical quantities that could be found in various disciplines we have encountered.

- SI units Publication guidelines

Testing

The Code for the units library is put through a series of CI tests before being merged. Current tests include running on CI systems.

Unit Tests

The units library has a series of units tests that are executed as part of the CI builds and when developing. They are built using Google test module. The module is downloaded if the tests are built. It uses Release 1.10 from the google test github repository

1. *examples_test* A simple executable that loads up some different types of measurements and does a few checks directly, it is mainly to test linking and has some useful features for helping with the code coverage measures.
2. *fuzz_issue_tests* tests a set of past fuzzing failures, including, errors, glitches, timeouts, round trip failures, and some round trip failures with particular flags.
3. *test_all_unit_base* **DO NOT RUN THIS TEST** it will take a very long time it does an exhaustive test of all possible unit bases to make sure the string conversion round trip works. I haven't actually executed it all yet.
4. *test_commodities* Run test using the commodity related functions and operations on precise_unit's
5. *test_conversions1* a series of tests about specific conversions, such as temperature, SI prefixes, extended SI units, and some other general operations about conversions
6. *test_conversions2* run through a series of test units and conversion from one of the converter websites, there are number of files that get used that contain known conversions
7. *test_equation_units* direct testing of the established equation units
8. *test_leadingNumbers* run a bunch of checks on the leading number processing for units and measurements, convert a leading string into a numerical value
9. *test_measurement* a series of tests on measurement objects including operations and comparisons, and construction
10. *test_measurement_strings* a few tests on the basic to and from string operations for measurements
11. *test_pu* tests of pu units and conversions
12. *test_random_round_trip* randomly pick a few 32 bit number, assume they are a unit and do a string conversion and interpretation on them and make sure they produce the same thing
13. *test_ucum* a series of tests coming from UCUM the units library tries to handle all official strings and a majority of the full names, and aliases
14. *test_udunits* a series of tests and test files coming from UDUNITS-2 Not all the units convert, some never will since they are ambiguous but we will probably allow a few more over time
15. *test_uncertain_measurements* test uncertain measurement operations using examples taken from web sources
16. *test_unit_ops* test operations on units, including mathematical expressions and comparison operators
17. *test_unit_strings* Unit strings test conversion to and from strings
18. *test_defined_units* tests checking all the unit string maps for duplicates and conflicts and correct sizing
19. *test_google_units* run some checks to ensure support for many units supported by google unit translation
20. *test_math* run some tests on the extra mathematical operations found in *units_math.hpp*
21. *test_siunits* run some tests of SI specific units and prefixes

CI systems

Azure

1. GCC 7.4 C++14 (Azure native Linux) 1. GCC 7.4 C++14 (Azure native Linux) with shared library build 1. AppleClang 11.0 (Xcode 11.3) C++17 1. AppleClang 11.0 (Xcode 11.3) C++11 1. MSVC 2019 C++17 1. MSVC 2019 C++11 1. MSVC 2022 C++20 1. GCC 4.8 C++11 1. GCC 7 C++11 1. GCC 7 C++14 1. GCC 8 C++17 1. GCC 9 C++17 1. GCC 12 C++20 1. Clang 3.4 C++11 1. Clang 3.5 C++11 1. Clang 8 C++14 1. Clang 9 C++17 1. Clang 14 C++20 1. Clang-tidy (both main library and tests)

Circle-CI

1. Clang 14, Thread Sanitizer
2. Clang 14, Address, undefined behavior sanitizer
3. Clang 14, Memory Sanitizer
4. Clang 8, Fuzzing library – run a couple of defined fuzzing tests from scratch to check for any anomalous situations. There are currently two fuzzers, the first test the units_from_string, and the second tests the measurement_from string. It first converts the fuzzing sequence, then if it is a valid sequence, converts it to a string, then converts that string back to a measurement or unit and makes sure the two measurements or units are identical. Any string sequence which doesn't work is captured and tested.

GitHub Actions

1. CodeQl 1. Coverage (ubuntu 22.04 image C++11, C++14, C++17, C++20, 32 and 64 bit unit base) 1. CPPLINT 1. Quick CMAKE checks for all supported versions of cmake

Codecov

Try to maintain the library at 100% coverage.

Pre-commit

Runs clang-format and many other checks for repo cleanliness

Sources of Unit String Definitions

The string processing tests and strings supported came from a number of different sources.

Converter App

As a simple example and potentially useful tool we made the converter app. It is a command line application that can be built as part of the units library to convert units given on the command line.

```
$ ./unit_convert 10 m ft
32.8084

$ ./unit_convert ten meters per second mph
22.3694

$ ./unit_convert --full ten meters per second mph
ten meters per second = 22.3694 mph

$ ./unit_convert --simplified ten meters per second miles/hour
10 m/s = 22.3694 mph

$ ./unit_convert -s four hundred seventy-three kilograms per hour pounds/min
473 kg/hr = 17.3798 lb/min

$ ./unit_convert -s 22 british fathoms *
10 british fathoms = 18.288 m
```

basically there are two options *–full,-f* and *–simplified,-s* a measurement which will take an arbitrary number of strings and a final string as a unit to convert to. It outputs the conversion and if specified the surrounding measurement and units either simplified or in the original. Using * or <*base*> in place of the unit string will result in converting the measurement to base units.

```
$ ./unit_convert --help
application to perform a conversion of a value from one unit to another
Usage: unit_convert [OPTIONS] measure... convert

Positionals:
  measure [TEXT ...] ... REQUIRED
                          measurement to convert .e.g '57.4 m', 'two thousand GB' '45.
↪7*22.2 feet^3/s^2'
  convert TEXT REQUIRED       the units to convert the measurement to

Options:
  -h,--help                   Print this help message and exit
  -f,--full                   specify that the output should include the measurement and␣
↪units
  -s,--simplified             simplify the units using the units library to_string␣
↪functions and print the conversion string like full. This option will take precedence␣
↪over --full
  --measurement [TEXT ...] ... REQUIRED
                          measurement to convert .e.g '57.4 m', 'two thousand GB' '45.
↪7*22.2 feet^3/s^2'
  --convert TEXT REQUIRED     the units to convert the measurement to
```

CHAPTER

TWO

INSTALLATION AND LINKING

The units library supports a header only mode and a compiled mode. One of the strengths of the library is the string processing which is only available in the compiled mode. Other additions from the compiled mode are the root operations on units and measurements. The header only mode includes the unit and measurement classes conversions between them and the definition library.

2.1 Header Only Use

The header only portion of the library can simply be copied and used. There are 3 headers *units_decl.hpp* declares the underlying classes. *unit_defintions.hpp* declares constants for many of the units, and *units.hpp* which is the primary public interface to units. If *units.hpp* is included in another file and the variable *UNITS_HEADER_ONLY* is defined then none of the functions that require the cpp files are defined. These header files can simply be included in your project and used with no additional building required. The *UNITS_HEADER_ONLY* definition is needed otherwise linking errors will result.

2.2 Compiled Usage

The second part is a few cpp files that can add some additional functionality. The primary additions from the cpp file are an ability to take roots of units and measurements and convert to and from strings. These files can be built as a standalone static library, a shared library or an object library, or included in the source code of whatever project want to use them. The code should build with an C++11 compiler. C++14 is recommended if possible to allow some additional function to be *constexpr*. Most of the library is tagged with constexpr so can be run at compile time to link units that are known at compile time. General Unit numerical conversions are not at compile time, so will have a run-time cost. A *quick_convert* function is available to do simple conversions. with a requirement that the units have the same base and not be an equation unit. The cpp code also includes some functions for commodities and will eventually have r20 and x12 conversions, though this is not complete yet.

2.3 Standalone Library

The units library can be built as a standalone library with either the static or shared library and installed like a typical package.

2.3.1 Unit Library CMake Reference

There are a few CMake variables that control the build process, they can be altered to change how the units library is built and what exactly is built.

CMake variables

- *BUILD_TESTING* : Generate CMake Variable controlling whether to build the tests or not
- *UNITS_ENABLE_TESTS* : Does the same thing as *BUILD_TESTING*
- *UNITS_BUILD_STATIC_LIBRARY*: Controls whether a static library should be built or not
- *UNITS_BUILD_SHARED_LIBRARY*: Controls whether to build a shared library or not, only one or none of *UNITS_BUILD_STATIC_LIBRARY* and *UNITS_BUILD_SHARED_LIBRARY* can be enabled at one time.
- *BUILD_SHARED_LIBS*: Controls the defaults for the previous two options, overriding them takes precedence
- *UNITS_BUILD_FUZZ_TARGETS*: If set to *ON*, the library will try to compile the fuzzing targets for clang libFuzzer
- *UNITS_BUILD_WEB_SERVER*: If set to *ON*, build a webserver, This uses boost::beast and requires boost 1.70 or greater to build it also requires CMake 3.12 or greater
- *UNITS_BUILD_CONVERTER_APP*: enables building a simple command line converter application that can convert units from the command line
- *UNITS_ENABLE_EXTRA_COMPILER_WARNINGS*: Turn on bunch of extra compiler warnings, on by default
- *UNITS_ENABLE_ERROR_ON_WARNINGS*: Mostly useful in some testing contexts but will turn on *Werror* so any normal warnings generate an error.
- *CMAKE_CXX_STANDARD*: Compile with a particular C++ standard, valid values are *11*, *14*, *17*, *20*, and likely *23* though that isn't broadly supported. Will set to *14* by default if not otherwise specified
- *UNITS_BINARY_ONLY_INSTALL*: Just install shared libraries and executables, no headers or static libs or packaging information
- *UNITS_CLANG_TIDY*: Enable the clang tidy tests as part of the build
- *UNITS_CLANG_TIDY_OPTIONS*: options that get passed to clang tidy when enabled
- *UNITS_BASE_TYPE*: Set to *uint64_t* for expanded base-unit power support. This increases the size of a unit by 4 Bytes.
- *UNITS_DOMAIN*: Specify a default domain to use for string conversions. Can be either a name from the domains namespace such as *domains::surveying* or one of 'COOKING', 'ASTRONOMY', 'NUCLEAR', 'SURVEYING', 'USE_CUSTOMARY', 'CLIMATE', or 'UCUM'.
- *UNITS_DEFAULT_MATCH_FLAGS*: Specify an integer value for the default match flags to be used for conversion
- *UNITS_DISABLE_NON_ENGLISH_UNITS*: the library includes a number of non-english units that can be converted from strings, these can be disabled by setting *UNITS_DISABLE_NON_ENGLISH_UNITS* to ON or setting the definition in the C++ code.
- *UNITS_DISABLE_EXTRA_UNIT_STANDARDS*: If set to *ON* disables UN recommendation 12, X12(not implemented yet), DOD(not implemented yet), from being included in the compilation and generated from strings.
- *UNITS_NAMESPACE*: The top level namespace of the library, defaults to *units*. When compiling with C++17 (or higher), this can be set to, e.g., *mynamespace::units* to avoid name clashes with other libraries defining *units*.

- *UNITS_INSTALL*: This is set to *ON* normally but defaults to *OFF* if used as a subproject. This controls whether anything gets installed by the install target.
- *UNITS_CMAKE_PROJECT_NAME*: This is set to *UNITS* by default. If using this in a package manager or wish to rename the project this variable can be set to another name to change the name of the package. This will change the install path and cmake target names. For example setting *-DUNITS_CMAKE_PROJECT_NAME=LLNL-UNITS* will create cmake project llnl-units::units, and llnl-units::header_only and will install in a llnl-units directory with appropriate cmake files.

If compiling as part of a subproject then a few other options are useful

- *UNITS_HEADER_ONLY*: Only generate the header only target, sets *UNITS_BUILD_STATIC_LIBRARY* and *UNITS_BUILD_SHARED_LIBRARY* to OFF
- *UNITS_INSTALL*: enable the install instructions of the library
- *UNITS_BUILD_OBJECT_LIBRARY*: Generate an object library that can be used as part of other builds. Only one of *UNITS_BUILD_SHARED_LIBRARY*, *UNITS_BUILD_STATIC_LIBRARY*, or *UNITS_BUILD_OBJECT_LIBRARY* can be set to *ON*. If more than one are set, the shared library and object library settings take precedence over the static library.
- *UNITS_LIBRARY_EXPORT_COMMAND*: If desired the targets for the units library can be merged into an root project target list by modifying this variable. The use cases for this are rare, but if this is something you want to do this variable should be set to something like *EXPORT rootProjectTargets*. It defaults to *"EXPORT unitsTargets"*

CMake Targets

If you are using the library as a submodule or importing the package there are a couple targets that can be used depending on the build. NOTE: these can be changed using *UNITS_CMAKE_PROJECT_NAME*.

- *units::units* will be set to the library being built, either the shared, static, or object
- *units::header_only* is a target for the headers if *UNITS_HEADER_ONLY* CMake variable is set, then only this target is generated. This target is always created.

Example

As part of the HELICS library the units library is used as a submodule it is included by the following code

```
# so units cpp exports to the correct target export
set(UNITS_INSTALL OFF CACHE INTERNAL "")

if(NOT CMAKE_CXX_STANDARD)
     set(CMAKE_CXX_STANDARD 17) # Supported values are ``11``, ``14``, and ``17``.
endif()

set(UNITS_BUILD_OBJECT_LIBRARY OFF CACHE INTERNAL "")
set(UNITS_BUILD_STATIC_LIBRARY ON CACHE INTERNAL "")
set(UNITS_BUILD_SHARED_LIBRARY OFF CACHE INTERNAL "")
set(UNITS_BUILD_CONVERTER_APP OFF CACHE INTERNAL "")
set(UNITS_BUILD_WEBSERVER OFF CACHE INTERNAL "")
set(UNITS_CLANG_TIDY_OPTIONS "" CACHE INTERNAL "")
set(UNITS_BUILD_FUZZ_TARGETS OFF CACHE INTERNAL "")

add_subdirectory(
     "${PROJECT_SOURCE_DIR}/ThirdParty/units" "${PROJECT_BINARY_DIR}/ThirdParty/units"
```

(continues on next page)

(continued from previous page)

```
)

set_target_properties(units PROPERTIES FOLDER Extern)

hide_variable(UNITS_HEADER_ONLY)
hide_variable(UNITS_BUILD_OBJECT_LIBRARY)
hide_variable(UNITS_NAMESPACE)
```

Then the target linked by

```
target_link_libraries(helics_common PUBLIC HELICS::utilities units::units)
```

CHAPTER

THREE

USER GUIDE

The Units library user guide is an in depth look at how to use the C++ library and its functionality, covering the basic types in the library and operations with them. The guide covers the basic types and what operations are available on them, as well as a lot of details on how to use the library.

3.1 Units

3.1.1 Basic Unit Types

There are two basic units classes *units* and *precise_units* They both include a *units_base* see *Unit base class* for the details. *units* has a single precision floating point multiplier and the *units_base* object. The *precise_unit* type uses a double precision floating point multiplier and includes commodity. The commodity is represented by a 32-bit code. See *Commodities* for more details on how that is used and defined. The simplest way to start is by using one of the *Defined Units* All standard units are defined and many non-standard ones as well.

The Basics of units are the seven SI base units:

- the kilogram (kg), for mass.
- the second (s), for time.
- the kelvin (K), for temperature.
- the ampere (A), for electric current.
- the mole (mol), for the amount of a substance.
- the candela (cd), for luminous intensity.
- the meter (m), for distance.

In addition to the base SI units a couple additional bases are defined:

- radian(rad), for angular measurement
- Currency ($), for monetary values
- Count (cnt), for single object counting

Currency may seem like a unusual choice in units but numbers involving prices are encountered often enough in various disciplines that it is useful to include as part of a unit. Technically count and radians are not units, they are representations of real things. A radian is a representation of rotation around a circle and is therefore distinct from a true unitless quantity even though there are no physical measurements associated with either. And count and mole are theoretically equivalent though as a practical matter using moles for counts of things is a bit odd for example *1 GB* of data is *~1.6605*10^-15 mol* of data. So they are used in different context and don't mix very often, the convert functions can convert between them if necessary.

The structure also defines some flags:

- *per-unit*, indicating per unit units
- *i_flag*, general flag and complex quantity
- *e_flag*, general unit discriminant
- *equation*, indicator that the unit is an equation unit.

Derived Units

A vast majority of physical units can be constructed using these bases, as well as many non-physical units. The entire structure for the units fits into 4 bytes to meet the design requirement for a compact type. This required a detailed evaluation of what physical units and combinations of them were in use in different scientific and commercial disciplines, The following list represents the range of allowed values chosen as the representation and those required by known and observed physical quantities.

- meter:[-8,+7] :normal range [-4,+4], intermediate ops [-6,+6]
- kilogram:[-4,+3] :normal range [-1,+1], intermediate ops [-2,+2]
- second:[-8,+7] :normal range [-4,+4], intermediate ops [-6,+6]
- ampere:[-4,+3] :normal range [-2,+2]
- kelvin:[-4,+3] :normal range [-4,+1]
- mole:[-2,+1] :normal range [-1,+1]
- candela:[-2,+1] :normal range [-1,+1]
- currency:[-2,+1] :normal range [-1,+1]
- count:[-2,+1] :normal range [-1,+1]
- radians:[-4,+3] :normal range [-2,+2]

For example the kilogram is rarely used in a squared context, so it has a normal range of between -1 and 1. But in intermediate mathematical operations it is squared on occasion, so we needed to be able represent that without overflow. Since without getting extraordinary complex we are limited to whole bit representation that infers a two's complement notation of 2 bits is [-2,-1] for 3 bits [-4,+3], and for 4 bits [-8,+7]. So for kilograms 3 bits were used. The pu flag was determined to be required by the initial design considerations, and a flag value also turned out to be required by library design requirements. The equation and e_flag flags came a little later in the library development but turned out to be very useful in representing other kinds of units and discriminating between some units.

3.1.2 Basic operations

Some mathematical operations between units are supported. * and / with units produce a new unit.

```
auto new_unit=m/s;
auto another=new_unit*s;
//another == m
```

produces a *new_unit* equivalent to meters/second.

3.1.3 Comparison Operators

Units also support the comparison operators ==, and *!*=.The other comparison operators are not supported as it is somewhat undefined whether m > kg or many other comparison like that. The inequality is the inverse of equality, but the equality operator is an interesting subject. The unit component is relatively straightforward that part is the base units, if those are not equivalent then the answer is false. However there is a floating point component to the unit representing a multiplier. And floating point equality is treacherous. What is done is a rounding operation with a range. Basically *units* are assuming to have 6 decimal digits of precision, while *precise_units* have 13. So units will result in equality as long as the first X significant digits in the multiplier are equivalent and the *unit_base* is equal. this can't be a specific range since the power of the multiplier is wide ranging this parsecs to picometers and all the base of meters.

3.1.4 Methods

Frequently units need to raised to some power. Units have a *pow(int)* method to accomplish this.

```
auto area_unit=m.pow(2);
```

The ^ will not work due to precedence rules in C++. If an operator for '^' were defined an operation such as m/s^2 would produce meters squared per second squared which is probably not what is expected. Therefore best not to define the operator and use a function instead.

Special Units

There are a few defined units that are special in some fashion, and can be used as sentinel values or have special operations associated with them.

Default Unit

The *defunit* unit is allowed to be converted to any other unit. it is equivalent of *per-unit*i_flag* The main use case is in the convert functions and makes a good

Error Unit

```
auto error_unit=unit(detail::unit_data(nullptr));
```

Invalid Unit

An invalid unit is any unit that is either the error unit or has a NaN in the multiplier. This is the unit returned from a string conversion if the string does not describe a unit or measurement.

one

The default constructor for *unit* and *precise_unit* is empty unit data and 1.0 in the multiplier.

There are also precise versions of these values in the *precise* namespace

Custom Units

The units library defines 1023 special custom units. These are custom units intended to specify a specific type of unit which doesn't have a normal unit base definition. The key idea behind the custom units is that they can be multiplied, divided by some normal powers of distance, mass, or time units and can be inverted

In strings these can be represented by "CXUN[X]" Where X is some number between 0 and 1023.

In C++ code they can be generated by

```
precise_unit new_cxc_unit=generate_custom_unit(code);
```

A set of checks and queries is available to check for custom_units.

- bool precise::custom::is_custom_unit(detail::unit_data udata);
- bool precise::custom::is_custom_unit_inverted(detail::unit_data udata);
- unsigned short precise::custom::custom_unit_number(detail::unit_data udata);

These checks will operate regardless of any m/kg/s unit combination or inverted units.

Custom units in Use

there are a few custom count units in use for specific clinical units Many of these units defy conversion to other known units but are used in pharmacological contexts So there is no translation to other units and cannot be converted except to multiple of the same unit. There are often well established tests for these units but no good way to convert them to other units. Many of these units come from UCUM.

- custom_unit(37): is hounsfield units used it CT and radiology
- custom_unit(49): is *erlang* used in telephone carrying capacity <https://en.wikipedia.org/wiki/Erlang_(unit)>`_
- many units in UCUM are defined like *[MPL'U]* or *[mclg'U]* for this context they define some unit which doesn't interact with other units in any known fashion. The notion used in the units library for string translations is that these define custom units. Rather than individually defining them, the library takes a hash of the part of the unit coming before the *'U]'* and generates a 10 bit hash. That 10 bit hash is used as the custom code for the units.
- custom_unit(77): is global warming potential related to climate operations
- custom_unit(78): is global temperature change potential

The other custom units are available for use or the one with known definition can be use if there is no domain conflicts.

The primary usage of these is for units that are procedurally defined and often used in the context of per mass or per volume or per time.

Implementation Details

Custom units use a combination of nearly all the different fields in the unit_base class, with the exception of count and radians. Based on the definitions used the custom units can be taken as per length/area/volume/mass/second with no issues. Some of the unit fields are used for defining an index and others are used purely for identification purposes.

Equation Units

The use of an equation flag in the unit_base defines a set of equation units. These are specific units where the relationship with other units is defined through an equation rather than a specific multiplier. There are 31 available equation units. Equation units use up the flags, count, and radian fields. All other units are left alone for defining the underlying units of the equation unit. So the equation specifier defines and equation rather than a specific unit. equation types 0-15 deal with logarithms in some way, 16-31 are undefined or represent some common scale type units

to extract the equation type - unsigned short precise::custom::eq_type(detail::unit_data udata);

Current equation definitions

- 0: log10(x)
- 1: nepers
- 2: bels
- 3: decibels
- 4: -log10(x)
- 5: -log10(x)/2.0
- 6: -log10(x)/3.0
- 7: -log10(x)/log10(50000)
- 8: log2(x)
- 9: ln(x)
- 10: log10(x)
- 11: 10*log10(x)
- 12: 2*log10(x)
- 13: 20*log10(x)
- 14: log10(x)/log10(3)
- 15: 0.5*ln(x)
- 16: API Gravity
- 17: Degree Baume Light
- 18: Degree Baume Heavy
- 19: UNDEFINED
- 20: UNDEFINED
- 21: UNDEFINED
- 22: saffir-simpson hurricane wind scale
- 23: Beaufort wind scale

- 24: Fujita scale
- 25: UNDEFINED
- 26: UNDEFINED
- 27: Prism diopter-100.0*tan(x)
- 28: UNDEFINED
- 29: Moment magnitude scale for earthquakes (richter)
- 30: Energy magnitude scale for earthquakes
- 31: UNDEFINED

The wind scales are not very accurate since they match up a slightly fuzzier notion to actual wind speed. There are general charts and the equations in use utilize a polynomial to approximate them to a continuous scale. So the units when used are generally convertible to a velocity unit such as m/s. There are currently 10 undefined equation units available if needed. The density scales (API Gravity, Degree Baume Heavy and Light) are based on handbook definitions at typical temperature scales. A few more like this might be added in the near future. Equation Value conversions ———————— The actual definitions of the equations are found in the *unit::precise::equation* namespace. Two functions are provided that convert values from equation values to units and vice versa.

- *double convert_equnit_to_value(double val, detail::unit_data UT)*
- *double convert_value_to_equnit(double val, detail::unit_data UT)*

also since some equation unit definitions depend on whether the actual units are power or magnitude values, there is a helper function to help determine this. *bool is_power_unit(detail::unit_data UT)* This applies in the neper, bel, and decibel units.

Custom Counting Units

The units library defines 16 special counting units. These are custom counting units intended to specify a specific type of event. The key idea behind the custom counting units is that they can be multiplied, divided by any powers of distance, mass, currency, or time units and can be inverted. The primary usage of these is for units that are procedurally defined and often used in the context of per mass or per volume or per time or per $.

In strings these can be represented by “CXCUN[X]” Where X is some number between 0 and 15.

In C++ code they can be generated by

```
precise_unit new_cxc_unit=generate_custom_count_unit(code);
```

A set of checks and queries is available to check for custom_count_units.

- bool precise::custom::is_custom_count_unit(detail::unit_data udata);
- bool precise::custom::is_custom_count_unit_inverted(detail::unit_data udata);
- unsigned short precise::custom::custom_count_unit_number(detail::unit_data udata);

These checks will operate regardless of any m/kg/s unit combination or inverted units. Underlying this is a set of codes and unit powers that would be extremely odd to encounter in normal use.

Custom count units in Use

there are a few custom count units in use for specific clinical units Many of these units defy conversion to other known units but are used in pharmacological contexts So there is no translation to other units and cannot be converted except to multiple of the same unit. There are often well established tests for these units but no good way to convert them to other units. Many of these units come from UCUM.

- custom_count_unit(0): is used for specific count units with commodities of some kind for string translation
- custom_count_unit(1): is *Arbitrary Unit* which has a clinical definition of some kind
- custom_count_unit(2): is International Unit
- custom_count_unit(3): is Index of reactivity which has a clinical definition
- custom_count_unit(4): is limit of flocculation which has a clinical definition
- custom_count_unit(5): is HPF or High Power field which is related to microscopy
- 6-15 are not currently in use.

The other custom units are available for use or the one with known definition can be use if there is no domain conflicts.

Implementation details

Custom count units utilizes the flags, candela, ampere, and Kelvin fields to make use of some non-physical unit definitions for a more useful purpose.

3.2 Measurements

The combination of a value and unit is known as a measurement. In the units library they are constructed by multiplying or dividing a unit by a numerical value.

```
measurement meas=10.0*m;
measurement meas2=5.3/s;
```

They can also be constructed via the constructor

```
measurement meas(10.0, kg);
measurement meas2(2.7, MW);
```

There are two kinds of measurements the regular *measurement* which uses a double precision floating point for the value and a *precise_measurement* which uses a double and a *precise_unit*. In terms of size the *measurement* class is 16 Bytes and the *precise_measurement* is 24 bytes.

3.2.1 Precise measurements

A precise measurement includes a double for the value and a precise_unit to represent the unit. Most of the string conversion routines to measurement produce a precise_measurement. the *measurement_cast* operation will convert a precise_measurement into a regular measurements.

```
precise_measurement mp(10.0, precise::kg);
measurement meas2=measurement_cast(mp);
```

3.3 Fixed Measurements

The primary difference between *fixed_measurement* and *measurement* is the idea that in a *fixed_measurement* the unit part is a constant. It does not change. Therefore any addition or subtraction operation will produce another measurement with the same units. It also allows for interaction and comparisons with numerical types since the unit is known. This is unlike measurements where comparison and addition and subtraction operations with numbers are not allowed. Otherwise the behavior and operations are identical between *measurement* and *fixed_measurement* and likewise between *fixed_precise_measurement* and *precise_measurement*

3.3.1 Relationship with numbers

Because the *unit* associated with a fixed measurement is fixed. It becomes legitimate to work with singular real valued numbers.

```
fixed_measurement dist(10, m);

if (dist>10.0) //this has meaning because the units of dist is known.
{
    //all other operators are defined with doubles
}

dist=dist+3.0;  // dist is now 13 meters

dist-=2.0;  // dist is now 11 meters

dist=5.0;  // dist is now 5 meters
```

3.3.2 Interactions with *measurement*

Fixed measurements have an implicit conversion to *Measurements*, so all the methods that work with measurement work with *fixed_measurements*. The construction of a fixed_measurement from measurement is explicit. Likewise fixed_precise_measurement` have an implicit conversion to *precise_measurement*, so all the methods that work with *precise_measurement* work with *fixed_precise_measurements*. The construction of a fixed_measurement from measurement is explicit.

3.4 Uncertain Measurements

The units library supports a class of measurements including an uncertainty measurement.

For Example *3.0±0.2m* would indicate a measurement of 3.0 meters with an uncertainty of 0.2 m.

All operations are supported The propagation of uncertainty follow the root sum of squares methods outlined Here. There are methods available such as *simple_divide*, *simple_product*, *simple_sum* and *simple_subtract* that just sum the uncertainties. The method in use in the regular operators assume that the measurements used in the mathematical operation are independent, and should use the sum of squares methods. A more thorough explanation can be found at this location.

The structure of an uncertain measurement consists of a float for the measurement value and a float for the uncertainty, and *unit* for the unit of the measurement.

3.4.1 Constructors

There are a number of different constructors for an uncertain measurement aimed at specify the uncertainty and measurement in different ways.

- *constexpr uncertain_measurement()* default constructor with 0 values for the value and uncertainty and a one for the unit.
- *constexpr uncertain_measurement(<float|double> val, <float|double> uncertainty, unit base)* : specify the parameters with values.
- *constexpr uncertain_measurement(<float|double> val, unit base)*: Just specify the value and unit, assume 0.0 uncertainty.
- *constexpr uncertain_measurement(measurement val, float uncertainty) noexcept* : construct from a measurement and uncertainty value.
- *uncertain_measurement(measurement val, measurement uncertainty) noexcept*: construct from a measurement value and uncertainty measurement. The uncertainty is converted to the same units as the value measurement.

3.4.2 Additional operators

Beyond the operations used in *Measurements*, there are some specific functions related to getting and setting the uncertainty.

- *uncertain_measurement& uncertainty(<double|float> newUncertainty)* : Will set the uncertainty value as a numerical value.
- *uncertain_measurement& uncertainty(const measurement &newUncerrtainty)*: will set the uncertainty as a measurement in specific units.
- *double uncertainty()*: Will get the current numerical value of the uncertainty
- *double uncertainty_as(units)*: will get the value of the uncertainty in specific units.
- *float uncertainty_f()*: will get the value of the uncertainty as a single precision floating point value.
- *constexpr measurement uncertainty_measurement()*: will return a measurement containing the uncertainty.
- *double fractional_uncertainty()*: will get the fractional uncertainty value. which is uncertainty/|value|.

3.4.3 String operations

The units library has some functions to extract an *uncertain_measurement* from a string - *uncertain_measurement_from_string(const std::string &ustring, std::uint64_t match_flags=0)*

The from string operation searches for an uncertainty marker then splits the string into two parts. It then uses the measurement from string operation on both halves of the string and forms an uncertain measurement from them depending on whether both halves have units and or values. Allowed uncertainty marker strings include [“+/-”, “±”, “±”, “+-”, “<u>+</u>”, “±”, “±”, “ \pm “]. These possibilities include unicode and ascii values and some sequences used in latex and html.

For Example all the following string will produce the same *uncertain_measurement*

- “3.1±0.3 m/s”
- “3.1 +/- 0.3 m/s”
- “3.1 ± 0.3 m/s”
- “3.1 m/s ±0.3 m/s”
- “3.1 m/s ±0.3”
- “3.1 meters per second ±0.3 m/s”
- “3.1 m/s +- 0.3*60 meters per minute”
- “3.1(3) m/s”

The last form is known as concise notation. The match flags are the same as would be used for converting *Measurements*

3.5 Units From Strings

The units library contains a few functions to generate various types from string representations

- *precise_unit unit_from_string(const std::string& ustring, std::uint32_t flags=0)* : will generate a precise_unit based on the data in the string
- *unit unit_cast_from_string(const std::string& ustring, std::uint32_t flags=0)*: will generate a unit based on the data in the string
- *precise_measurement measurement_from_string(const std::string& ustring, std::uint32_t flags=0)*: will generate a precise_measurement from the data in the string
- *measurement measurement_cast_from_string(const std::string& ustring, std::uint32_t flags=0)*: will generate a measurement from the data in the string
- *uncertain_measurement uncertain_measurement_from_string(const std::string& ustring, std::uint32_t flags=0)*: will generate an uncertain_measurement from the data in the string

The general form is to take a string and optionally a flag object. See *Conversion Flags* for a detailed description of the flags. Generally it is fine to leave off the flag argument.

3.5.1 Unit Strings

in general the unit string conversion is intended to be as flexible as possible. just about any unit normally written can be converted.

For example

- “m/s”
- “meter/second”
- “meter/s”
- “metre/s”
- “meters/s”
- “meters per second”
- “metres per second”
- “m per sec”
- “meterpersecond”
- “METERS/s”
- “m*s^-1”
- “meters*seconds^(-1)”
- “(second/meter)^(-1)”
- “100 centimeters / 1000 ms”

Will all produce the unit of meters per second. As a note there are quite a few more units that can be converted from strings than are listed in the *Defined Units*. Numbers are supported and become part of the unit. “99 feet” would create a new unit with a definition of 99 ft. The multiplier stored would include the conversion from meters to feet*99. This allows for arbitrary unit definitions. The + operator also works if the units on both sides have the same base for example *3 ft + 2 in* would be the equivalent of *38 inches*. If the units do not have the same base + is interpreted as a multiplication.

3.5.2 Measurement strings

The conversion from a string to measurement looks for a leading number before the unit. The “99 feet” in the previous example would then get a measurement value of 99 and the unit would be feet. The measurement from string function also can interpret written numbers such as “three thousand four hundred and twenty-seven miles” This should get correctly read as 3427 miles.

The conversion function also handles a few cases where the unit symbol is written before the value such as currency *$27.92* would be a value of 27.92 with the currency unit.

3.5.3 Uncertain Measurements

Similarly to Measurement strings, uncertain measurements can also be converted from strings see *Uncertain Measurements* for additional details on the formats supported.

3.6 Units To Strings

All the class in the units library can be given as an argument to a *to_string* function. This function converts the units or value into a *std::string* that is representative of the unit or measurement. In all cases the primary aim of the *to_string* is to generate a string that the corresponding **_from_string* function will recognize and convert back to the original unit. The secondary aim is to generate string that is human readable in standard notation. While this is achieved for many common units there is some work left to do to make it better.

For example

```
measurement density=10.0*kg/m.pow(3);
measurement meas2=2.7*puMW;

auto str1=to_string(density);
auto str2=to_string(meas2);

// from google tests
EXPECT_EQ(str1, "10 kg/m^3");
EXPECT_EQ(str2, "2.7 puMW");
```

```
uncertain_measurement um1(10.0, 0.4, m);
auto str = to_string(um1);
EXPECT_EQ(str, "10+/-0.4 m");
```

Uncertain measurement string conversions make some attempt to honor significant digits based on the uncertainty.

3.6.1 Advanced Usage

The *to_string* function also takes a second argument which is a *std::uint64_t match_flags* in all cases this default to 0, it is currently unused though will be used in the future to allow some fine tuning of the output in specific cases. In the near future a flag to allow utf 8 output strings will convert certain units to more common utf8 symbols such as unit Powers and degree symbols, and a few others. The output string would default to ascii only characters.

3.6.2 Stream Operators

Output stream operators are NOT included in the library. It was debatable to include them or not but there would be a lot of additional overloads that would add quite a bit of code to the header files, that in most cases is not necessary so the decision was made to exclude them. The *to_string* operations provide most of the capabilities with some additional flexibility, and if needed for a particular use case can be added to the user code in a simple fashion

```
namespace units{
   std::ostream& operator<<(std::ostream& os, const precise_unit& u)
   {
       os << to_string(u);
       return os;
```

(continues on next page)

(continued from previous page)

```
    }
} // namespace units
```

Any of the types in the units library with a *to_string* operation can be handled in the same way. Depending on the compiler, placing the operator in the namespace may or may not be necessary.

3.6.3 Underlying Conversion Map Access

The underlying conversion maps may be accessed by users if desired. To access them a compile time definition needs to be added to the build *ENABLE_UNIT_MAP_ACCESS*

```
#ifdef ENABLE_UNIT_MAP_ACCESS
namespace detail {
    const std::unordered_map<std::string, precise_unit>& getUnitStringMap();
    const std::unordered_map<unit, const char*>& getUnitNameMap();
}
#endif
```

These may be useful for building a GUI or smart lookup or some other operations. *getUnitStringMap()* returns a map of known unit strings, and *getUnitNameMap()* is a mapping of common units back to strings as a building block for the *to_string* operation.

3.7 Math Operations

Additional mathematical operations on measurements are available in the *unit_math.hpp* header these are header only so no additional compilation is required. The intention of this header is to match operations from the *cmath* header available in the standard library. These are all template functions which will work for any measurement type.

3.7.1 Type traits

The header includes a few type traits used in the header file and potentially useful elsewhere including

- *is_unit* : true if the type is a unit (unit or precise_unit)
- *is_measurement* : true if the type is one of the defined measurement types
- *is_precise_measurement* : true if the type is one of the defined precise measurement type

3.7.2 Rounding and Truncation

These operations will effect only the value part of the measurement

- *round*
- *trunc*
- *ceil*
- *floor*

3.7.3 Trigonometric functions

Trigonometric operations will operate only if the measurement is convertible to radians

- *sin*
- *cos*
- *tan*

3.7.4 Multiplies and divides

Division and multiplication operators for measurements that have support for per_unit measurement

- *multiplies* : works like * except when one of the measurements is per_unit and they have the same unit base, then they remove the per unit
- *divides* : works like / except if both measurements have the same base then the result has a *per_unit* unit

See *Strain* for examples on usage

3.7.5 Others

Other common mathematical expressions found in <*cmath*>

- *fmod* : return the floating point modulus of a division operation as long as division is a valid operation
- *hypot* : works for two and three measurements or floating point values as long as addition is a valid operation.
- *cbrt* : works similarly to the *sqrt* operation

3.8 Commodities

the *precise_unit* class can represent commodities as well as units. The commodity is represented by a 32 bit unsigned number that codes a variety of commodities. The actual representation is still undergoing some change so expect this to change going forward. See *Commodity Details* for more details.

3.8.1 Methods

There are a few available methods for dealing with commodity codes and string translation

- *std::uint32_t getCommodity(std::string comm)* - will get a commodity from a string.
- *std::string getCommodityName(std::uint32_t commodity)* - will translate a commodity code into a string

Custom Commodities

- *void addCustomCommodity(std::string comm, std::uint32_t code)* - Add a custom commodity code using a string and code
- *void clearCustomCommodities()* - clear all current user defined commodities
- *void disableCustomCommodities()* - Turn off the use of custom commodities
- *void enableCustomCommodities()* - Turn on the ability to add and check custom commodities for later access

3.8.2 Commodities to names

The *getCommodityName* methods has 4 stages and will return from any successful stage.

1. Check custom commodities if allowed
2. Check standardized commodity names
3. Check for special codes for name storage (short names <=5 ascii lower case characters are stored directly in the code)
4. generate string "CXCOMM[<code>]"

3.8.3 String To Commodities

The *getCommodity* method works nearly the opposite of *getCommodityName*.

1. Check custom commodities if allowed
2. Check standardized commodity codes
3. Check for string "CXCOMM[<code>]"
4. Check for special codes for name storage (short names <=5 ascii lower case characters are stored directly in the code)
5. Generate a hash code of the string and if allowed store it as a custom commodity

3.8.4 Defined Commodities

The list of commodities is still in development. Generally **`traded commodities<https://en.wikipedia.org/wiki/List_of_traded_commodities>`_** are available as well as a few others that are used in clinical definitions or other uses as part of unit definition standards. In the future this list will more generally expand to match international trade tables. See **`commodities.cpp<https://github.com/LLNL/units/blob/master/units/commodities.cpp>`_** for details on the exact list.

3.9 User defined units

The units library has support for user defined units and *Commodities*. These interact with the **_from_string* and *to_string* operations to allow custom conversions and definitions.

3.9.1 Defining a custom unit

The basic function for adding a custom unit is *addUserDefinedUnit(string name, precise_unit un)*

For example from a test in the library

```
precise_unit idgit(4.754, mol/m.pow(2));
addUserDefinedUnit("idgit", idgit);

auto ipm=unit_from_string("idgit/min");
EXPECT_EQ(ipm, idgit / min);

auto str = to_string(ipm);
EXPECT_EQ(str, "idgit/min");

str = to_string(ipm.inv());
EXPECT_EQ(str, "min/idgit");
```

Basically user defined units can interact with the string conversion functions just like any other unit defined in the library. A user defined unit gets priority when converting to a string as well including when squared or cubed as part of a compound unit. For example from the test cases:

```
addUserDefinedUnit("angstrom", units::precise::distance::angstrom);

auto str = to_string(units::unit_from_string("us / angstrom^2"));
EXPECT_EQ(str, "us/angstrom^2");
str = to_string(units::unit_from_string("us / angstrom"));
EXPECT_EQ(str, "us/angstrom");
```

If only an ability to interpret strings is needed the *addUserDefinedInputUnit* can be used

```
precise_unit idgit(4.754, mol/m.pow(2));
addUserDefinedInputUnit("idgit", idgit);

auto ipm = unit_from_string("idgit/min");
EXPECT_EQ(ipm, idgit / min);

auto str = to_string(ipm);
EXPECT_EQ(str.find("idgit"), std::string::npos);
EXPECT_NE(str.find("kat") , std::string::npos);
```

If only output strings are needed the *addUserDefinedOutputUnit* can be used

```
precise_unit idgit(4.754, mol / m.pow(2));
addUserDefinedOutputUnit("idgit", idgit);

auto ipm = unit_from_string("idgit/min");
//this is not able to be read since idgit is undefined as an input
```

(continues on next page)

(continued from previous page)

```
EXPECT_NE(ipm, idgit / min);

auto str = to_string(idgit/min);
/** output only should make this work*/
EXPECT_EQ(str,"idgit/min");
```

The output unit can be used when the interpreter works fine but the string output doesn't do what you want it to do.

A unit can be removed from the user defined unit set via *removeUserDefinedUnit*

```
 auto ipm=unit_from_string("idgit/min");
EXPECT_EQ(ipm, idgit / min);

auto str = to_string(ipm);
EXPECT_EQ(str, "idgit/min");

str = to_string(ipm.inv());
EXPECT_EQ(str, "min/idgit");

removeUserDefinedUnit("idgit");
EXPECT_FALSE(is_valid(unit_from_string("idgit/min")));
```

The removal also works for user defined units specified via *addUserDefinedInputUnit* or *addUserDefinedOutputUnit*

3.9.2 Input File

Sometimes it is useful to have a larger library of units in this case the *std::string definedUnitsFromFile(const std::string& filename)* can be used to load a number of units at once.

The file format is quite simple. # at the beginning of a line indicates a comment other wise

```
# comment
meeter == meter
meh == meeter per hour
# => indicates input only unit
     mehmeh => meh/s
# <= indicates output only unit
     hemhem => s/meh
```

or

```
# comment
yodles=73 counts

# comment
"yeedles", 19 yodles

yimdles; dozen yeedles
```

or

```
# test the quotes for inclusion
"bl==p"=18.7 cups
```

(continues on next page)

(continued from previous page)

```
# test single quotes for inclusion
'y,,p',9 tons

# ignore just one quote
'np==14 kg

# escaped quotes
"j\"\""= 13.5 W

# escaped quotes
'q""'= 15.5 W
```

The basic rule is that one of *[<=,;]* will separate a definition name from a unit definition. If the next character after the separator is an '=' it is ignored. If it is a '>' it implies input only definition. If the separator is an '<=' then it is output only. Otherwise it calls *addUserDefinedUnit* for each definition. The function is declared *noexcept* and will return a string with each error separated by a newline. So if the result string is *empty()* there were no errors.

3.9.3 Other Library Operations

- *clearUserDefinedUnits()* will erase all previously defined units
- *disableUserDefinedUnits()* will disable the use of user defined units
- *enableUserDefinedUnits()* will enable their use if they had been disabled, they are enabled by default.

3.9.4 Notes on units and threads

The user defined units usage flag is an atomic variable but the modification of the user defined library are not thread safe, so if threads are needed make all the changes in one thread before using it in other threads, or protect the calls with a separate mutex. The disable and enable functions trigger an atomic variable that enables the use of user defined units in the string translation functions. disableUserDefinedUnits() also turns off the ability to specify new user defined units but does not erase those already defined.

3.10 Physical constants

The units library comes with a number of physical constants with appropriate units defined. All the physical constants are specified as *Precise measurements* and in the namespace *units::constants* In general the most recent definition was chosen which includes the 2019 redefinition of some SI units this matches with the rest of the library and the defined units. Inspiration for the different constants was taken from wikipedia and NIST. Defined constants. The 2019 redefinition of the SI system was used where applicable. All common constants listed from NIST are included

3.10.1 Defined constants

Values are taken from NIST 2018 CODATA unless otherwise noted

- Standard gravity - *g0*
- Gravitational constant - *G*
- Speed of light - *c*
- Elementary Charge (2019 redefinition) - *e*
- hyperfine structure transition frequency of the caesium-133 atom - *fCs*
- fine structure constant - *alpha*
- Planck constant (2019 redefinition) - *h*
- Reduced Planck constant (2019 redefinition) - *hbar*
- Boltzman constant (2019 redefinition) - *k*
- Avogadros constant (2019 redefinition) - *Na*
- Luminous efficiency - *kcd*
- Permittivity of free space - *eps0*
- Permeability of free space - *mu0*
- Gas Constant - *R*
- Stephan Boltzmann constant -*s*
- Hubble constant 69.8 km/s/Mpc - *H0*
- Mass of an electron - *me*
- Mass of a proton - *mp*
- Bohr Radius - *a0*
- Faraday's constant - *F*
- Atomic mass constant - *mu*
- Conductance quantum - *G0*
- Josephson constant - *Kj*
- Magnetic flux quantum - *phi0*
- von Kiltzing constant - *Rk*
- Rydberg constant - *Rinf*

3.10.2 Planck Units

These units are found in the *units::constants::planck* namespace and include *length*, *mass*, *time*, *charge*, and *temperature*.

3.10.3 Atomic units

These physical constants are values related to an electron or atomic measurements They include *length*, *mass*, *time*, *charge*- same as *e* above, *energy*, and *action*. The atomic constants are defined in the *units::constants::atomic* namespace.

3.10.4 Numbers

There are a few numbers that are used in the library and include definitions in the *units::constants* namespace. They are represented as doubles and are defined as constexpr

- *pi* (3.14159265358979323846)
- *tau* (2.0*pi)
- *invalid_conversion* (signaling NaN)
- *infinity*
- *standard_gravity* the numerical value of g0, earth standard gravity in m/s/sec
- *speed_of_light* The numerical value of the speed of light in m/s

The last two are used in several other units and some conversions so it seemed better to just define the numerical value and use that rather than use the same number in several places.

3.10.5 Planetary masses

The masses of some of the solar system bodies are included in *units::constants::Planetary::mass*

- solar
- earth
- moon
- jupiter
- mars

3.10.6 From Strings

All constants listed here are available for conversion from strings by wrapping in brackets For example the luminous efficiency would be converted to a unit by using *[kcd]* The planck constants are available as *[planck::XXXXX]* or *planckXXXXXX and the atomic constants are available as `[atomic::XXXX]*

3.10.7 Uncertainties

Certain physical constants have uncertainties associated with them and have an additional *uncertain_measurement* associated with them see uncertain_measurments. These can be found in the *units::constants::uncertain* namespace and include:

- Gravitational constant - *G*
- Permittivity of free space - *eps0*
- Permeability of free space - *u0*

- Hubble constant 69.8 km/s/Mpc - *H0*
- Mass of an electron - *me*
- Mass of a proton - *mp*
- Atomic mass constant - *mu*
- mass of nuetron - *mn*
- Rydberg constant - *Rinf*
- fine structure constant - *alpha*

NOTE: A few of the uncertain constants have more precision than supported in uncertain_measurments but were included for completeness

3.11 Defined Units

The units library comes with two sets of units predefined. They are all located in src/units/unit_definitions.hpp. The definitions come in two flavors a precise_unit and a regular unit. All the precise units are defined in the namespace *units::precise*

All the units are defined as a constexpr values. The choice of which units to define is somewhat arbitrary and guided by the authors experience and the origins of the library in power systems and electrical engineering in the US. Units that the author has actually encountered in work or life are included and in cases where there might be conflicts depending on the location preference was given to the US customary definition, though international systems take priority.

3.11.1 Listing of Units

The most common units are defined in the namespace *units::precise* and others are defined in subnamespaces.

Base Units

Most base units have two definitions that are equivalent

- meter, m
- kilogram, kg
- second, s
- Ampere, A
- Kelvin, K
- mol
- candela, cd
- currency
- count
- pu
- iflag
- eflag
- radian, rad

Specialized units

Some specialized units are defined for use in conversion applications or for making handling string conversions a little easier

- defunit - special unit that signifies conversion to any other units is possible
- invalid - special unit that conversion has failed
- error - an error unit

Numerical Units

Sometimes it is useful to have pure numerical units, often for multiplication with other units such as *hundred*kg* or something like that which becomes a single unit with 100 kg.

- one
- ten
- hundred
- percent (0.01)
- infinite
- nan

Also included in this category are SI prefixes deci, centi,milli, micro, nano, pico, femto, atto, zepto, yocto, ronto, quecto, deka, hecto, kilo, mega, giga, tera, peta, exa, hecto, zetta, yotta,rotta, quetta.

and SI data prefixes kibi, mebi, gibi, tebi, pebi, exbi, zebi, yobi

Derived SI Units

There are many units that used in combination with the SI system that are derived from the base units

- hertz, Hz
- volt, V
- Pa (pascal, on some systems this is defined as something else so the definition(*pascal*) is skipped)
- newton, N
- joule, J
- watt, W
- coulomb, C
- farad, F
- siemens, S
- weber, Wb
- tesla, T
- henry, H
- lumen, lm
- lux, lx

- becquerel, Bq
- gray, Gy
- sievert, Sv
- katal, kat
- sr

Extra SI related units

A few units that are not officially part of the SI but are accepted for use with the SI system, along with a few other SI units with prefixes that are commonly used.

- mg
- g
- mL
- L
- nm
- mm
- km
- cm
- bar

Centimeter-Gram-Second system

The CGS system is a variant on the metric system. Units from the CGS system are included under the namespace *units::precise::cgs*.

- eng
- dyn
- barye
- gal
- poise
- stokes
- kayser
- oersted
- gauss
- debye
- maxwell
- biot
- gilbert
- stilb

- lambert
- phot
- curie
- roentgen
- REM
- RAD
- emu
- langley
- unitpole
- statC_charge
- statC_flux
- abOhm
- abFarad
- abHenry
- abVolt
- statV
- statT
- statHenry
- statOhm
- statFarad

Conventional Electrical Units

defined in namespace *units::precise::conventional*

- volt90
- ampere90
- watt90
- henry90
- coulomb90
- farad90
- ohm90

Meter Gram Force System

defined in namespace *units::precise::gm*

- pond
- hyl
- at
- poncelet
- PS

Meter Tonne Second system

Defined in namespace *units::precise::MTS*

- sthene
- pieze
- thermie

Additional Time units

Defined in namespace *units::precise::time*, units marked with * are also defined in the *units::precise*.

- min*
- ms*
- ns*
- hr*
- h*
- day*
- week
- yr* (8760 hr)
- fortnight
- sday - sidereal day
- syr - sidereal year
- at - mean tropical year
- aj - mean julian year
- ag - mean gregorian year
- year - aliased to median calendar year (365 days) which is the standard for SI
- mos - synodal (lunar) month
- moj - mean julian month
- mog - mean gregorian month

International customary Units

These are traditional units that have some level of international definition Defined in namespace *units::precise::i*

- grain
- point
- pica
- inch
- foot
- yard
- mile
- league
- hand
- cord
- board_foot
- mil
- circ_mil

A few units have short symbols defined in *unit::precise* in, ft, yd, mile. These alias to the international definition.

Avoirdupois units

Avoirdupois units are another common international standard of units for weight and volumes. The units are defined in *units::precise::av*

- dram
- ounce
- pound
- hundredweight
- longhundredweight
- ton
- longton
- stone
- lbf
- ozf
- slug
- poundal

A few common units have symbols defined in *units::precise* lb, ton, oz, lbf and these alias to the Avoirdupois equivalent.

Troy Units

Most commonly for precious metals a few units are defined in *units::precise::troy*, with a basis in the international grain.

- pennyweight
- oz
- pound

United States Customary Units

These are traditional units defined in the United States, for survey or common usage. Defined in *unit::precise::us*.

- foot
- inch
- mil
- yard
- rod
- chain
- link
- furlong
- mile
- league
- acre*
- homestead
- section
- township
- minim
- dram
- floz
- tbsp
- tsp
- pinch
- dash
- shot
- gill
- cup
- pint
- quart
- gallon

- flbarrel - liquid barrel
- barrel
- hogshead
- cord
- fifth

A few US customary units are defined in specific namespaces to distinguish them from other forms US customary dry measurements are defined in *units::precise::us::dry*

- pint
- quart
- gallon
- peck
- bushel
- barrel
- sack
- strike

Some grain measures used in markets and commodities are defined in *units::precise::us::grain*. When commodities are a little more developed this will be defined with appropriate commodity included.

- bushel_corn
- bushel_wheat
- bushel_barley
- bushel_oats

Some survey units are defined in *units::precise::us::engineers* to distinguish them from others

- chain
- link

The unit gal (gallon) is also defined in *units::precise* since that is pretty common to use.

FDA and metric measures

The food and drug administration has defined some customary units in metric terms for use in medicine. These are defined in *units::precise::metric* Also included are some other customary units that have a metric definition.

- tbsp
- tsp
- floz
- cup
- cup_uslegal
- carat

Canadian Units

Some Canadian definitions of customary units defined in *units::precise::canada*

- tbsp
- tsp
- cup
- cup_trad
- gallon
- grain::bushel_oats

Australia Units

Traditional Australian units defined in *units::precise::australia*

- tbsp
- tsp
- cup

Imperial or British Units

Traditional british or imperial units, defined in *units::precise::imp*.

- inch
- foot
- thou
- barleycorn
- rod
- chain
- link
- pace
- yard
- furlong
- league
- mile
- nautical_mile
- knot
- acre
- perch
- rood
- gallon

- quart
- pint
- gill
- cup
- floz
- tbsp
- tsp
- barrel
- peck
- bushel
- dram
- minim
- drachm
- stone
- hundredweight
- ton
- slug

Apothecaries System

Used in pharmaceutical contexts the apothecaries system of units is defined in *units::precise::apothecaries*.

- floz (same as imperial version)
- minim
- scruple
- drachm
- ounce
- pound
- pint
- gallon
- metric_ounce

Nautical Units

Some units defined in context of marine travel defined in *units::precise::nautical*

- fathom
- cable
- mile
- knot
- league

Japanese traditional Units

Some traditional Japanese units are included for historical interest in *units::precise::japan*

- shaku
- sun
- ken
- tsubo
- sho
- kan
- go
- cup

Chinese Traditional Units

Some traditional Chinese units are included for historical interest in *units::precise::chinese*

- jin
- liang
- qian
- li
- cun
- chi
- zhang

Typographic units

Units used in typesetting and typography are included in *units::precise::typographic* Subsets of the units depending on the location are in subnamespaces

units::precise::typographic::american

- line
- point
- pica
- twip

units::precise::typographic::printers

- point
- pica

units::precise::typographic::french

- point
- ligne
- pouce
- didot
- cicero
- pied
- toise

units::precise::typographic::metric

- point
- quart

units::precise::typographic::IN

l'Imprimerie nationale

- point
- pica

units::precise::typographic::tex

- point
- pica

units::precise::typographic::postscript

- point
- pica

units::precise::typographic::dtp

This is the modern standard or as close to such a thing as exists

- point
- pica
- twip
- line

units::precise::typographic

Values taken from dtp namespace

- point
- pica

Distance Units

Some additional distance units are defined in *units::precise::distance*

- ly
- au
- au_old
- angstrom
- parsec
- *smoot*
- cubit
- longcubit
- arpent_us
- arpent_fr
- xu

Area Units

Some additional units defining an area *units::precise::area*

- are
- hectare
- barn
- arpent

Mass Units

Some additional units defining a mass *units::precise::mass*

- quintal
- ton_assay
- longton_assay
- Da
- u
- tonne

t is included in the *units::precise* namespace as a metric tonne

Volume Units

Some additional units defining a volume *units::precise::volume*

- stere
- acre_foot
- drum

Angle Units

A few units defining angles are defined in *units::precise::angle*.

- deg*
- gon
- grad
- arcmin
- arcsec
- brad - binary radian (1/256 of a circle)

Directional Units

A few cardinal directional units are defined in *units::precise::direction*, these make use of the *i_flag* and a numerical value

- east
- north
- south
- west

Temperature Units

A few units related to temperature systems, defined in *units::precise::temperature*

- celsius, degC*
- fahrenheit, degF*
- rankine, degR
- reaumur

Pressure Units

Some units related to pressure are defined in *units::precise::pressure*

- psi
- inHg
- mmHg
- torr
- inH2O
- mmH2O
- atm - standard atmosphere
- att - technical atmosphere

Power Units

Some units related to power are defined in *units::precise::power*

- hpE - electric Horsepower
- hpI - international horsepower
- hpS - steam horesepower
- hpM - mechanical horsepower

the unit hp is aliased in the *units::precise* namespace to *power::hpI*

Energy Units

Some units related to energy are defined in *units::precise::energy*

- kWh
- MWh
- eV
- kcal
- cal_4 - calorie at 4 deg C
- cal_15 - calorie at 15 deg C
- cal_28 - calorie at 28 deg C
- cal_mean - mean calorie
- cal_it - international table calorie
- cal_th - thermochemical calorie
- btu_th - thermochemical BTU
- btu_39 - BTU at 39 deg C
- btu_59 - BTU at 59 deg C
- btu_60 - BTU at 60 deg C
- btu_mean - mean BTU
- btu_it - international table BTU
- btu_iso - rounded btu_it
- quad
- tonc - cooling ton
- therm_us
- therm_br
- therm_ec
- EER - energy efficiency ratio
- SG - specific gravity
- ton_tnt
- boe - barrel of oil equivalent
- foeb
- hartree
- tonhour
- tce - ton of coal equivalent
- lge - liter of gasoline equivalent

in the *units::precise* namespace

- btu = energy::but_it
- cal = energy::cal_th

- kWh = energy::kWh
- MWh = energy::MWh

Power system Units

Some additional units related to power systems and electrical engineering in *units::precise::electrical* namespace

- MW
- VAR - W*i_flag
- MVAR
- kW
- kVAR
- mW
- puMW
- puV
- puHz
- MJ
- puOhm
- puA
- kV
- mV
- mA

Equation type Units

Equation units are explained more thoroughly in *Equation Units* Some of the specific common equation units are defined in the namespace *units::precise::log*.

- neper
- logE - natural logarithm
- neperA - neper of amplitude unit
- neperP - neper of a power unit
- logbase10
- bel
- belP - bel of a power based unit
- dBP
- belA - bel of an amplitude based unit
- dBA - dB of an amplitude based unit
- logbase2
- dB

- neglog10
- neglog100
- neglog1000
- neglog50000
- B_SPL
- B_V
- B_mV
- B_uV
- B_10nV
- B_W
- B_kW
- dB_SPL
- dB_V
- dB_mV
- dB_uV
- dB_10nV
- dB_W
- dB_kW
- dB_Z - radar reflectivity
- BZ - radar reflectivity

Textile related Units

Units related to the textile industry in namespace *units::precise::textile*.

- tex
- denier
- span
- finger
- nail

Clinical Units

Units related to clinical medicine in namespace *units::precise::clinical*.

- pru
- woodu
- diopter
- prism_diopter
- mesh

- charriere
- drop
- met
- hounsfield

Laboratory Units

Units used in laboratory settings in namespace *units::precise::laboratory*.

- svedberg
- HPF
- LPF
- enzyme_unit
- IU
- arbU - arbitrary unit
- IR - index of reactivity
- Lf - Limit of flocculation
- PFU
- pH
- molarity
- molality

Data Units

Units related to computer data and storage in *units::precise::data*

- bit*
- nibble
- byte
- kB*
- MB*
- GB*
- kiB
- MiB
- GiB
- bit_s - Shannon bit for information theory
- shannon
- hartley
- ban
- dit

- deciban
- nat
- trit
- digits

B is defined as byte in *units::precise*

Computation units

Units related to computation *units::precise::computation*.

- flop
- flops
- mips

Special units

Some special units that were not otherwise characterized in namespace *units::precise::special*.

- ASD - amplitude spectral density
- moment_magnitude - moment magnitude for earthquake scales (related to richter scale)
- moment_energy
- sshws - saffir simpson hurricane wind scale
- beaufort - Beaufort wind scale
- fujita - Fujita scale for tornados
- mach - mach number(multiplier of the speed of sound)
- rootHertz - square root of Hertz, this is a special handling unit that triggers some specific behavior to handle it.
- rootMeter - square root of meter, this is a special handling unit that triggers some specific behavior to handle it.
- degreeAPI - API scale for measuring liquid density typically petroleum based products
- degreeBaumeLight - scale for measuring liquid density for liquids lighter than water
- debreeBaumeHeavy - scale for measuring liquid density for liquids heavier than water

Other Units

General purpose other units not otherwise categorical in namespace *units::precise::other*

- ppm - part per million
- ppb - part per billion
- candle
- faraday
- rpm* - revolution per minute
- CFM - cubic feet per minute
- MegaBuck - $1,000,000

- GigaBuck - $1,000,000,000

Climate Units

Units related to climate in namespace *units::precise::climate*

- gwp - global warming potential
- gtp - global temperature potential
- odp - ozone depletion unit

Speed Units

mph and mps are defined in *units::precise* since they are pretty common

3.11.2 Units in the *units* namespace

Regular units are defined in the *units* namespace. The general rule is that any units with a definition directly in *units::precise* has an analog *nonprecise* unit in the *units* namespace. One addition is that any unit defined in *precise::electrical* also is defined in *units* this has to do with the origins of the library in power systems.

3.12 Unit Domains

There are some ambiguous unit symbols. Different fields use the same symbol to mean different things. In the units library the definition has defaulted to SI standard definition with two known ambiguities. the symbol 'a' is used for *are*, the symbol *rad* refers to radians.

However there are occasions where the units from one field or another are desired. The units library applies the notion of a unit domain which can be passed to the unit_flags argument for any string conversion, for a few select units this will change the resulting from a string.

3.12.1 Available Domains

thus far 5 specific unit domains have been defined they are in the *units::domains* namespace.

- *ucum* – THE UNIFIED CODE FOR UNITS OF MEASURE
- *cooking* – units and symbols commonly used for recipes
- *astronomy* – units and symbols used in astronomy
- *nuclear* – units and symbols used in nuclear or particle physics
- *surveying* – units and symbols used in surveying in the United states
- *us_customary* – units and symbols traditionally used in the us(combination of cooking and surveying)
- *climate* – units and symbols used in climate science
- *allDomains* – this domain does all the above domains where not mutually exclusive. So mostly a combination of ucum and astronomy/nuclear with a few us customary units IT is not recommended to use this but provided for the combinations

The only units and symbols using the domain are those that might be ambiguous or contradictory to the ST definition. The specific units affected are defined in the next section.

3.12.2 Domain Specific Units

These are unit definitions affected by specifying a specific unit domain

UCUM

- *B* – bel vs Byte
- *a* – julian year vs are

Astronomy

- *am* – arc minute vs attometer
- *as* – arc second vs attosecond
- *year* – mean tropical year vs median calendar year

Cooking

- *C* – cup vs coulomb
- *T* – Tablespoon vs Tesla
- *c* – cup vs speed of light
- *t* – teaspoon vs metric tonne
- *TB* – Tablespoon vs TeraByte
- *smi* – smidgen vs square mile
- *scruple* – slightly different definition when used in cooking context
- *ds* – dessertspoon vs deci second

Surveying

- ‘ and all variants refer to feet vs arcmin
- ’’ and all variants refer to inches vs arcsec

Nuclear

- *rad* radiation absorbed dose vs radian
- *rd* same as *rad* vs rod

Climate

- *kt* kilo-tonne vs karat

US customary

Combination of surveying and cooking

All domains

Combination of all of the above

More units will likely be added to this as the need arises

3.12.3 Specifying the domain

The domain can be specified in the unit_flag string supplied to the *unit_from_string* operation.

```
auto unit1=units::unit_from_string(unitString,nuclear_units)
```

when used as part of the flags argument the definitions are in the *unit_conversion_flags* enumeration

- *strict_ucum*
- *cooking_units*
- *astronomy_units*
- *surveying_units*
- *nuclear_units*
- *climate_units*
- *us_customary_units*

A default domain can also be specified though

```
setUnitsDomain(code);
```

with the code using one of those found in the *units::domains* namespace. this domain will be used unless another is specified through the match flags. This function return the previous domain which can be used if only setting the value temporarily.

The default domain can be set at compile time through the *UNITS_DEFAULT_DOMAIN* definition

```
#define UNITS_DEFAULT_DOMAIN units::domains::astronomy
#include "units/units.hpp"
```

In CMake this field can be defined and will be directly translated. The *UNITS_DOMAIN* CMake variable can also be used to specify a domain as a string like *UCUM* or *COOKING* and have it appropriately translate. See *Unit Library CMake Reference* for more details.

3.13 Conversion Flags

The *units_from_string* and *to_string* operations take an optional flags argument. This controls certain aspects of the conversion process. In most cases this can be left to default unless very specific needs arise.

3.13.1 Unit_from_string flags

- *default_conversions* – no_flags, so using the default operations
- *case_insensitive* –perform case insensitive matching for UCUM case insensitive matching
- *single_slash* –specify that there is a single numerator and denominator only a single slash in the unit operations
- *strict_si* –input units are strict SI
- *strict_ucum* –input units are matching UCUM standard
- *cooking_units* –input units for cooking and recipes are prioritized
- *astronomy_units* –input units for astronomy are prioritized
- *surveying_units* –input units for surveying are prioritized
- *nuclear_units* –input units for nuclear physics and radiation are prioritized
- *climate_units* –input units for climate sciences
- *us_customary_units* – input units for us customary measurements are prioritized(same as *cooking_units* | *surveying_units*)
- *numbers_only* –indicate that only numbers should be matched in the first segments, mostly applies only to power operations
- *no_recursion* –don't recurse through the string
- *not_first_pass* –indicate that is not the first pass
- *no_per_operators* –skip matching "per"
- *no_locality_modifiers* –skip locality modifiers
- *no_of_operator* –skip dealing with "of"
- *no_addition* – skip trying unit addition
- *no_commodities* –skip commodity checks
- *skip_partition_check* –skip the partition check algorithm
- *skip_si_prefix_check* –skip checking for SI prefixes
- *skip_code_replacements* –don't do some code and sequence replacements
- *minimum_partition_size2* –specify that any unit partitions must be greater or equal to 2 characters
- *minimum_partition_size3* –specify that any unit partitions must be greater or equal to 3 characters
- *minimum_partition_size4* –specify that any unit partitions must be greater or equal to 4 characters
- *minimum_partition_size5* –specify that any unit partitions must be greater or equal to 5 characters
- *minimum_partition_size6* –specify that any unit partitions must be greater or equal to 6 characters
- *minimum_partition_size7* –specify that any unit partitions must be greater or equal to 7 characters

Indications for use

The *case_insensitive* flag should be used to ignore capitalization completely. It is targeted at the UCUM upper case specification but is effective for all situations where case should be ignored.

The library is by nature somewhat flexible in capitalization patterns, because of this some strings are allowed that otherwise would not be if SI were strictly followed. For example: *Um* would match to micro meters which should not if being exacting to the SI standard. The *strict_si* flag prevents some not all of these instances, and whether others can be disabled is being investigated.

The *single_slash* flag is targeted at a few specific programs which use the format of a single slash marking the separation of numerator from denominator.

strict_ucum, *cooking_units*, *astronomy_units*, *surveying_units*, *nuclear_units*, *climate_units* and *us_customary_units* are part of the domain system and can change the unit matched.

The remainder of the flags are somewhat self explanatory and are primarily used as part of the string conversion program to prevent infinite recursion. The *no_commodities* or *no_per_operator* may be used if it is known those do not apply for a slight increase in performance. The *no_recursion* or *skip_partition_check* can be use if only simple strings are passed to speed up the process somewhat.

The minimum partition size flags can be used to restrict how much partitioning it does which can reduce the possibility of "false positives" or unwanted unit matches on the strings

All the flags can be "or"ed to make combinations such as *minimum_partition_size4|astronomy_units|no_of_operator*

3.13.2 to_string Flags

- *disable_large_power_strings* - if the units definition allows large powers this flag can disable the use of them in the output string.

The to_string flags can be combined with the other conversion flags without issue.

Default flags

Flags will normally default to *0U* however they can be modified through *setDefaultFlags*. This function returns the previous value in case it is needed to swap them temporarily. The flags can be retrieved via *getDefaultFlags()* This function is automatically called if no flag argument is passed. The initial value can be set through a compile time or build time option *UNITS_DEFAULT_MATCH_FLAGS*.

CHAPTER

FOUR

APPLICATION NOTES

This folder is a collection of some example code snippets and discussions applied to various situations

4.1 Strain

Strain is an interesting unit in that it is a dimensionless unit. The most common expression is in *in/in* or *mm/mm* often defined as . The effects of strain are pretty small so µ is also pretty common. It is the fractional change in length. There are a several ways to represent this in the units library depending on the needs for a particular situation.

4.1.1 Method 1

The first way is simply as a ratio.

```
measurement deltaLength=0.00001*m;
measurement length=1*m;

auto strain=deltaLength/length;

EXPECT_EQ(to_string(strain), "1e-05");

//applied to a 10 ft bar
auto distortion=strain*(10*ft);
EXPECT_EQ(to_string(distortion), "0.0001 ft");
```

The main issue that there is no distinction between a strain measurement and any other ratio, but in many cases that is fine.

4.1.2 Method 2

The default defined unit of strain in the units library uses per unit meters as a basis. The multiplies and divides methods in the units math library can take per unit flag into account when doing the multiplication to get the original units back. The advantages of this are that strain becomes a distinctive unit from all other ratio units. Volumetric or area strain can be represented in the same way. It does have the disadvantage of requiring the *multiplies*

```
#include <units/units_math.hpp> //for multiplies
precise_measurement strain=1e-05*default_unit("strain");
EXPECT_EQ(to_string(strain), "1e-05 strain");
```

(continues on next page)

(continued from previous page)

```
//applied to a 10 ft bar
auto distortion=multiplies(strain,(10*ft));
EXPECT_EQ(to_string(distortion), "0.0001 ft");
```

```
#include <units/units_math.hpp> //for multiplies divides
measurement deltaLength=0.00001*m;
measurement length=1*m;

auto strain=divides(deltaLength,length);
EXPECT_EQ(to_string(strain), "1e-05 strain");

//applied to a 10 ft bar
auto distortion=multiplies(strain,(10*ft));
EXPECT_EQ(to_string(distortion), "0.0001 ft");
```

4.1.3 Method 3

The third potential method is to use one of the indicator flags to define a unit for strain. This can work in cases where there is no other potential conflicts with the flag and you need the * operator to work.

```
precise_unit ustrain(1e-6,eflag);  // microstrain

addUserDefinedUnit("ustrain",ustrain);
precise_measurement strain=45.7*ustrain;
EXPECT_EQ(to_string(strain), "45.7 ustrain");

//applied to a 10 m bar
auto distortion=strain*(10*m);
EXPECT_DOUBLE_EQ(distortion.value_as(precise::mm),0.457);
```

The advantages of this are that the there is no per unit values to handle. The disadvantage is that the eflag needs to be handled particularly when dealing with strings. If it is just dealing with computations this is less of an issue. So this method can work fine in some cases.

4.1.4 Discussion

There are several ways to represent strain or any other ratio unit that is derived from particular unit cancellations. All have advantages and disadvantages in particular situations and the method of choice will come down to the expected use cases. The library chooses the per unit method as it maintains the source units, but other choices are free to choose if they work better in particular situations.

CHAPTER

FIVE

THE LOW LEVEL DETAILS OF THE UNITS LIBRARY

5.1 Unit base class

The unit base class is a bitmap comprising segments of a 32 bit number. all the bits are defined

the underlying definition is a set of bit fields that cover a full 32 bit unsigned integer

```
// needs to be defined for the full 32 bits
     signed int meter_ : 4;
     signed int second_ : 4; // 8
     signed int kilogram_ : 3;
     signed int ampere_ : 3;
     signed int candela_ : 2; // 16
     signed int kelvin_ : 3;
     signed int mole_ : 2;
     signed int radians_ : 3; // 24
     signed int currency_ : 2;
     signed int count_ : 2; // 28
     unsigned int per_unit_ : 1;
     unsigned int i_flag_ : 1; // 30
     unsigned int e_flag_ : 1; //
     unsigned int equation_ : 1; // 32
```

The default constructor sets all the fields to 0. But this is private and only accessible from friend classes like units.

The main constructor looks like

```
constexpr unit_data(
        int meter,
        int kilogram,
        int second,
        int ampere,
        int kelvin,
        int mole,
        int candela,
        int currency,
        int count,
        int radians,
        unsigned int per_unit,
        unsigned int flag,
        unsigned int flag2,
        unsigned int equation)
```

an alternative constructor

```
explicit constexpr unit_data(std::nullptr_t);
```

sets all the fields to 1

5.1.1 Math operations

When multiplying two base units the powers are added. For the flags. The e_flag and i_flag are added, effectively an Xor while the pu and equation are ORed.

For division the units are subtracted, while the operations on the flags are the same.

Power and Root and Inv functions

For power operations all the individual powers of the base units are multiplied by the power number. The pu and equation flags are passed through. For even powers the i_flag and e_flag are set to 0, and odd powers they left as is. For root operations, First a check if the root is valid, if not the error unit is returned. If it is a valid root all the powers are divided by the root power. The Pu flag is left as is, the i_flag and e_flag are treated the same is in the pow function and the equations flag is set to 0.

There is one exception to the above rules. There is a special unit for Hz it is a combination of some i_flag and e_flag and a high power of the seconds unit. This unit is used in amplitude spectral density and comes up on occasion in some engineering contexts. There is some special logic in the power function that does the appropriate things such the square of Hz= Hz. If a low power of seconds is multiplied or divided by the special unit it still does the appropriate thing. But *Hz*Hz* will not generate the expected result. Hz is a singular unique unit and the only coordinated result is a power operation to remove it. Hz unit base itself uses a power of (-5) on the seconds and sets the i_flag and e_flag.

The inverse function is equivalent to pow(-1), and just inverts the unit_data.

5.1.2 Getters

The unit data type supports getters for the different fields all these are constexpr methods

- *meter()*: return the meter power
- *kg()*: return the kilogram power
- *second()*: return the seconds power
- *ampere()*: return the ampere power
- *kelvin()*: return the kelvin power
- *mole()*: return the mole power
- *candela()*: return the candela power
- *currency()*: return the currency power
- *count()*: return the count power
- *radian()*: return the radian power
- *is_per_unit()*: returns true if the unit_base has the per_unit flag set
- *is_equation()*: returns true if the unit_base has the equation field set
- *has_i_flag()*: returns true if the i_flag is set

- *has_e_flag()*: returns true if the e_flag is set
- *empty()*: will check if the unit_data has any of the base units set, flags are ignored.
- *unit_type_count*: will count the number of base units with a non-zero power

5.1.3 Modifiers

there are a few methods will generate a new unit based on an existing one the methods are constexpr

- *add_per_unit()*: will set the per_unit flag
- *add_i_flag()*: will set the i_flag
- *add_e_flag()*: will set the e_flag

The method *clear_flags* is the only non-const method that modifies a unit_data in place.

5.1.4 Comparisons

Unit data support the == and *!*= operators. these check all fields.

There are a few additional comparison functions that are also available.

- *equivalent_non_counting(unit_base other)* : will return true if all the units but the counting units are equal, the counting units are mole, radian, and count.
- *has_same_base(unit_base other)*: will return true if all the units bases are equivalent. So the flags can be different.

5.2 Commodity Details

The *precise_unit* class includes an unsigned 32-bit unsigned integer that represents a commodity of some kind.

This is a 32 bit code representing a commodity and possibly containers or form factor.

So while there is some predefined structure to the commodities. Any user is free to use it however they like as it can be manipulated as 32 bit code however a user might wish to use it. The conversion to and from string is governed by the following rules.

The high order bit(31) is a power, either 1 or -1. So a 1 in high bit represents an inverse commodity, for example a unit of *$/oz* of gold would have an inverse power of gold, while the *$/oz* would be in the *precise_unit*. Upon division all bits in the commodity are inverted.

5.2.1 Control code

bits 29 and 30 are control codes *00* is a normal commodity *01* is a normal commodity with form factor code *10* is a direct definitions *11* is a custom commodity defined in a map storage

Direct definitions

The direct definitions define a set of codes that are defined in a couple different methods

The next 3 bits define which method

000 short strings, 5 lower case characters+`_`+'{|}~' (ascii codes 95-126) *001* 3 byte alpha numeric code *010* 6 character hex code *011* 4 byte code ascii code 32-95 [numbers+upper case+punctuation] *100* short strings, 5 upper case characters+@[]^_' (ascii codes 64-95) *101* UNUSED *110* UNUSED *111* pure common commodity codes

others will be defined later.

Short Strings

To avoid always having to do a map lookup, many commodities or commodity codes can be represented by a short string of 5 or fewer characters. These cannot be case sensitive so '_' is a space or null character and if at the end of the string will be removed for display purposes. The very limited character set includes '_', *a-z',* '' and, '{|}~'. This is meant to simplify a chunk of the use cases. Custom Commodity Strings which are not captured in this mode fall into the custom commodity bin. The bits for this kind of commodity definition are 010000U[AAAAA][BBBBB][CCCCC][DDDDD][EEEEE], with A, B, C, D, and E representing the bits of the code letters. There are 2 codes one representing the lower case character set, and one with the upper cases character set with different punctuation marks. For the upper case set, setting the *U* bit to 1 indicates a stock symbol.

3 byte code

For short alpha/numeric codes of 3 bytes or fewer the byte code can be captured in the lower 24 bits of the commodity code. The bits for this kind of commodity definition are 010001[UU][AAAAAAAA][BBBBBBBB][CCCCCCCC], with A,B, C representing the bits of the code letters. The codes UU, define a set of types of code

00 user defined *01* UNDEFINED *10* ISO currency codes defined in ISO 4217 *11* UNDEFINED

6 character hex code

Similar to the 3 byte code some commodities can be represented by a 6 byte hex code

The bits for this kind of commodity definition are 010011XX[AAAA][BBBB][CCCC][DDDD][EEEE][FFFF], with A, B, C, D, E, F representing the bits of hex codes.

4 character codes

Similar to the 3 byte code some commodities can be represented by a 6 byte hex code

The bits for this kind of commodity definition are 010011[UU][AAAAAA][BBBBBB][CCCCCC][DDDDDD], with A, B, C, D, representing the characters

00 user defined *01* Chemical Formula *10* UNDEFINED *11* UNDEFINED

Known Definitions

A set of known commodities are defined in the header libraries. These are contained using code 111 and are defined in header files. The first 6 bits are defined 010111 leaving 26 bits available for user defined commodity codes.

Custom Commodity

String which can't be represented by the very simplistic short string mode are placed into a hash table for lookup and assigned a hash code generated from the string. The string is converted to a 29-bit hash placed in the lower 29 bits of the commodity code.

Normal Commodity with Form Factor

Frequently commodities come in a specific form factor. With a form factor code in place this can represent a form factor independent of the actual commodity material. For example a drum of oil vs a drum of gasoline.

the container is wrapped in a 8-bit code bits 21-28. The commodity itself is contained in bits 0-20. The bit codes for packaging is 001[FFFFFFFF][CCCCCCC][CCCCCCC][CCCCCCC]. To the extent possible the form factor codes in use are those used in recommendation 21 of international trade for use in conjunction with harmonized code. This covers the trade of goods but in general is insufficient to cover all the required packaging modes necessary for general description so it is not used exactly. The codes 0-99 if used correspond to codes used in recommendation 21. The way this is a encoded is the lowest 7 bits correspond to the recommendation if the value < 99 since that is a 2 digit decimal numerical code. Numbers 100-127 and 228-255 are local user definitions defined as required for other purposes. Numbers 128 to 227 correspond to alternate names for recommendation 21 codes, this is to disambiguate strings when converting to and from string representations. In Rec 21 codes 70-79 are reserved for future use but may be used in the units library as needed.

Normal Commodity

The codes used for normal commodity are the same as those used with a container with the exception that the additional 8 bit can be used for more specific codes of that commodity used for international trade. The codes used are based on the harmonized system for international trade <https://www.trade.gov/harmonized-system-hs-codes>`_ The 0-20 bits contain the harmonized system 6 digit code. The chapter is contained in bits 14-20, the section in bits 7-15, and the subsection in bits 0-6. This allows structure that can act as a mask on specific types of commodities. Common commodities are mapped to chapter and section mostly, though some exceptions go to the subsection for commodity to string translation. The 6 digit harmonized commodity code is the same between using with a container and without. If no container is used. the additional 8 bits, can represent the country specific codes.

In the normalized code 7 bit sections, the codes for 100-127 represent other commodities that cannot be represented in the allowable 8 bits of space. These are stored in a hash map when used for later reference as needed. This allows representation of a large percentage of codes with no additional overhead and an additional 5.6 million codes through the hash structure. This is far more codes than are presently in use.

5.3 Parsing of squared and cubic

When units are written there are a few terms that modify the powers of a unit. The two primary terms are *square* and *cubic*

These are rules the library follows when parsing terms such as this

- *square* or *sq* or *sq.* will apply to the unit immediately following the term
- *cubic* or *cu* or *cu.* will apply the unit immediately following the term
- *squared* will apply to the unit immediately preceding the term
- *cubed* will apply to the unit immediately preceding the term

CHAPTER

SIX

UNITS ON THE WEB

You can try out the string conversions through the units Webserver

This page allows you to enter a measurement string and a unit string for conversion.

The measurement string can be of any form with a number and units

- *10 m*
- *hundred pounds*
- *45.673 GB*
- *dozen feet*

the unit string should be some unit that is convertible from the measurement units:

- *inches*
- *troy oz*
- *kiB*
- *british fathoms*

The conversion also supports mathematical operations see *Units From Strings* for additional details on string conversions. The units can also be set to *`or `<base>* to convert the measurement to base units.

6.1 Rest API

The units web server does not serve files, it generates all responses on the fly. There are 3 URI indicators it responds to beyond the root page.

- */convert* : responds with an html page
- */convert_trivial* : responds with the results as a simple text
- */convert_json* : responds with a json string containing the requested conversions and the results.

For example in Linux or anything with curl

```
$ curl -s "13.52.135.81/convert_trivial?measurement=10%20tons&units=lb"
20000

$ curl -s "13.52.135.81/convert_json?measurement=10%20tons&units=lb"
{
"request_measurement":"",
"request_units":"lb"",
```

(continues on next page)

(continued from previous page)

```
"measurement":"",
"units":"lb"",
"value":"nan"
}

 $ curl -s "13.52.135.81/convert_json?measurement=ten%20meterspersecond&
↪units=feetperminute&caction=to_string"
{
"request_measurement":"ten meterspersecond",
"request_units":"feetperminute",
"measurement":"10 m/s",
"units":"ft/min",
"value":"1968.5"
}
```

This works with POST or GET methods. The *caction* field can be set to "to_string" this will "simplify" the units in the result or at least use the internal to_string operations to convert to an interpretable string in more accessible units.