The UDUNITS-2 C API

Steven R. Emmerson

Table of Contents

1 Synopsis

Coding:

```
#include <udunits2.h>
      ut_system*                ut_read_xml(const char* path);

      ut_system*                ut_new_system(void);

      void                      ut_free_system(ut_system* sys-
                                tem);

      ut_system*                ut_get_system(const ut_unit* unit);

      ut_unit*                  ut_get_dimensionless_unit_one(const ut_system* sys-
                                tem);

      ut_unit*                  ut_get_unit_by_name(const ut_system* sys-
                                tem, const char* name);

      ut_unit*                  ut_get_unit_by_symbol(const ut_system* sys-
                                tem, const char* symbol);

      ut_status                  ut_set_second(const ut_unit* sec-
                                ond);

      ut_status                  ut_add_name_prefix(ut_system* sys-
                                tem, const char* name, double value);

      ut_status                  ut_add_symbol_prefix(ut_system* sys-
                                tem, const char* symbol, double value);

      ut_unit*                  ut_new_base_unit(ut_system* sys-
                                tem);

      ut_unit*                  ut_new_dimensionless_unit(ut_system* sys-
                                tem);

      ut_unit*                  ut_clone(const ut_unit* unit);

      void                      ut_free(ut_unit* unit);

      const char*               ut_get_name(const ut_unit* unit, ut_encoding en-
                                coding);
```

```
ut_status               ut_map_name_to_unit(const char* name, const ut_encodi
                        coding, const ut_unit* unit);

ut_status               ut_unmap_name_to_unit(ut_system* sys-
                        tem, const char* name, const ut_encoding en-
                        coding);

ut_status               ut_map_unit_to_name(const ut_unit* unit, const char*
                        coding);

ut_status               ut_unmap_unit_to_name(const ut_unit* unit, ut_encodin
                        coding);

const char*             ut_get_symbol(const ut_unit* unit, ut_encoding en-
                        coding);

ut_status               ut_map_symbol_to_unit(const char* sym-
                        bol, const ut_encoding encoding, const ut_unit* unit)

ut_status               ut_unmap_symbol_to_unit(ut_system* sys-
                        tem, const char* symbol, const ut_encoding en-
                        coding);

ut_status               ut_map_unit_to_symbol(const ut_unit* unit, const char
                        bol, ut_encoding encoding);

ut_status               ut_unmap_unit_to_symbol(const ut_unit* unit, ut_encod
                        coding);

int                     ut_is_dimensionless(const ut_unit* unit);

int                     ut_same_system(const ut_unit* unit1, const ut_unit* u

int                     ut_compare(const ut_unit* unit1, const ut_unit* unit2

int                     ut_are_convertible(const ut_unit* unit1, const ut_uni

cv_converter*           ut_get_converter(ut_unit* from, ut_unit* to);

ut_unit*                ut_scale(double factor, const ut_unit* unit);

ut_unit*                ut_offset(const ut_unit* unit, dou-
                        ble offset);
```

```
ut_unit*                ut_offset_by_time(const ut_unit* unit, dou-
                        ble origin);

ut_unit*                ut_multiply(const ut_unit* unit1, const ut_unit* unit

ut_unit*                ut_invert(const ut_unit* unit);

ut_unit*                ut_divide(const ut_unit* numer, const ut_unit* de-
                        nom);

ut_unit*                ut_raise(const ut_unit* unit, int power);

ut_unit*                ut_root(const ut_unit* unit, int root);

ut_unit*                ut_log(double base, const ut_unit* ref-
                        erence);

ut_unit*                ut_parse(const ut_system* sys-
                        tem, const char* string, ut_encoding en-
                        coding);

char*                   ut_trim(char* string, ut_encoding en-
                        coding);

int                     ut_format(const ut_unit* unit, char* buf, size_t size
                        signed opts);

ut_status               ut_accept_visitor(const ut_unit* unit, const ut_visit
                        itor, void* arg);

double                  ut_encode_date(int year, int month, int day);

double                  ut_encode_clock(int hours, int min-
                        utes, double seconds);

double                  ut_encode_time(int year, int month, int day, int hou
                        ble second);

void                    ut_decode_time(double value, int* year, int* month, i
                        ble* second, double* resolution);

ut_status               ut_get_status(void);

void                    ut_set_status(ut_status status);
```

```
int                          ut_handle_error_message(const char* fmt, ...);

ut_error_message_handler     ut_set_error_message_handler(ut_error_message_handler
                            dler);

int                          ut_write_to_stderr(const char* fmt, va_list args);

int                          ut_ignore(const char* fmt, va_list args);



float                        cv_convert_float(const cv_converter* con-
                            verter, float value);

double                       cv_convert_double(const cv_converter* con-
                            verter, double value);

float*                       cv_convert_floats(const cv_converter* con-
                            verter, const float* in, size_t count, float* out);

double*                      cv_convert_doubles(const cv_converter* con-
                            verter, const double* const in, size_t count, dou-
                            ble* out);

void                         cv_free(cv_converter* conv);
```

Compiling:

```
c89 -I includedir ...
```

Where *includedir* is the installation-directory for C header files (e.g., `/usr/local/include`).

Linking:

```
c89 ... -L libdir -l udunits2 ... -l m
```

Where *libdir* is the installation-directory for object code libraries (e.g., `/usr/local/lib`).

2 What's a Unit Package Good For?

The existance of a software package is justified by what you can do with it. The three main things you can do with the UDUNIT-2 package are

1. Chapter 4 [Value Conversion], page 11.
2. Convert a string representation of a unit into a binary one — enabling the programatic manipulation of units. There are three ways to do this:
   - Section 3.2 [Extracting], page 7 from a [unit-system], page 5. This requires that you know the unit's name or symbol and that the unit is in a unit-system.
   - Chapter 5 [Parsing], page 13. This requires that the string be parsable by `[ut_parse()], page 13`.
   - Chapter 8 [Operations], page 19.
3. Chapter 7 [Formatting], page 18 — enabling the printing and storing of units in a human-readable form.

While the above might seem to be trivial activities, their general availability at the time might have helped prevent the Mars Climate Orbiter fiasco.

3 Unit-Systems

A unit-system is a set of units that are all defined in terms of the same set of base units. In the SI system of units, for example, the base units are the meter, kilogram, second, ampere, kelvin, mole, and candela. (For definitions of these base units, see http://physics.nist.gov/cuu/Units/current.html.)

In the UDUNITS-2 package, every accessible unit belongs to one and only one unit-system. It is not possible to convert numeric values between units of different unit-systems. Similarly, units belonging to different unit-systems always compare unequal.

There are several categories of operations on unit-systems:

3.1 Obtaining a Unit-System

Typically, you would obtain a unit-system of predefined units by reading the default unit database using [ut_read_xml(), page 6] with a NULL pathname argument. If this doesn't quite match your needs, then there are alternatives. Together with the typical solution, the means for obtaining a useful unit-system are (in order of increasing difficulty):

- Obtain the default unit-system using [ut_read_xml(), page 6](NULL).
- Copy and customize the unit database and then call [ut_read_xml(), page 6] with the pathname of the customized database to obtain a customized unit-system.
- Same as either of the above but then adding new units to the unit-system using [ut_new_base_unit(), page 8] and [ut_new_dimensionless_unit(), page 8].
- Same as the above but also deriving new units using Chapter 8 [Operations], page 19 and then adding them to the unit-system using Chapter 9 [Mapping], page 25.
- Same as the above but starting with an empty unit-system obtained from [ut_new_system(), page 7], in which case you will definitely have to start with [ut_new_base_unit(), page 8] and [ut_new_dimensionless_unit(), page 8].

You should pass every unit-system pointer to [ut_free_system(), page 9] when you no longer need the corresponding unit-system.

`ut_system* ut_read_xml (const char* path)` [Function]

Reads the XML-formatted unit-database specified by *path* and returns the corresponding unit-sytem. If *path* is NULL, then the pathname specified by the environment variable UDUNITS2_XML_PATH is used if set; otherwise, the compile-time pathname of the installed, default, unit database is used. You should pass the returned pointer to ut_free_system() when you no longer need the unit-system. If an error occurs, then this function writes an error-message using [ut_handle_error_message(), page 30] and returns NULL. Also, [ut_get_status(), page 29] will return one of the following:

UT_OPEN_ARG
: *path* is non-NULL but the file couldn't be opened. See errno for the reason.

UT_OPEN_ENV
: *path* is NULL and environment variable UDUNITS2_XML_PATH is set but the file couldn't be opened. See errno for the reason.

`UT_OPEN_DEFAULT`
: *path* is `NULL`, environment variable `UDUNITS2_XML_PATH` is unset, and the installed, default, unit database couldn't be opened. See `errno` for the reason.

`UT_OS`
: Operating-system error. See `errno`.

`UT_PARSE`
: The database file couldn't be parsed.

`ut_system* ut_new_system (void)` [Function]

Creates and returns a new unit-system. On success, the unit-system will be empty except for the dimensionless unit one. You should pass the returned pointer to `ut_free_system()` when you no longer need the unit-system. If an error occurs, then this function writes an error-message using [ut_handle_error_message()], page 30 and returns `NULL`. Also, [ut_get_status()], page 29 will return the following:

`UT_OS`
: Operating-system error. See `errno`.

3.2 Extracting Units from a Unit-System

NOTE: This section covers low-level access to the indidual units of a [unit-system], page 5. General parsing of arbitrary unit specifications is coverted in the section Chapter 5 [Parsing], page 13.

A [unit-system], page 5 contains mappings from identifiers to units (and vice versa). Consequently, once you have a unit-system, you can easily obtain a unit for which you know the name or symbol using the function [ut_get_unit_by_name()], page 7 or [ut_get_unit_by_symbol()], page 7.

`ut_unit* ut_get_unit_by_name (const ut_system* system, const char* name)` [Function]

Returns the unit to which *name* maps from the unit-system referenced by *system* or `NULL` if no such unit exists. Name comparisons are case-insensitive. If this function returns `NULL`, then [ut_get_status()], page 29 will return one of the following:

`UT_SUCCESS`
: *name* doesn't map to a unit of *system*.

`UT_BAD_ARG`
: *system* or *name* is `NULL`.

`ut_unit* ut_get_unit_by_symbol (const ut_system* system, const char* symbol)` [Function]

Returns the unit to which *symbol* maps from the unit-system referenced by *system* or `NULL` if no such unit exists. Symbol comparisons are case-sensitive. If this function returns `NULL`, then [ut_get_status()], page 29 will return one of the following:

`UT_SUCCESS`
: *symbol* doesn't map to a unit of *system*.

`UT_BAD_ARG`
: *system* or *symbol* is `NULL`.

`ut_unit* ut_get_dimensionless_unit_one (const ut_system* system)` [Function]

Returns the dimensionless unit one of the unit-system referenced by *system*. While not necessary, the returned pointer may be passed to `ut_free()` when you no longer need the unit. If *system* is `NULL`, then this function writes an error-message using [ut_handle_error_message(), page 30] and returns `NULL`. Also, [ut_get_status(), page 29] will return `UT_BAD_ARG`.

3.3 Adding Units to a Unit-System

If you use [ut_read_xml(), page 6], then you should not normally need to add any new units to a unit-system.

Because you get units via their names or symbols, adding a unit to a unit-system actually means mapping one or more identifiers (i.e., names or symbols) to the unit. Thereafter, you can use [ut_get_unit_by_name(), page 7] and [ut_get_unit_by_symbol(), page 7] to retrieve the unit. The mapping of identifiers to units is covered Chapter 9 [Mapping], page 25.

Having said that, it is possible to create a new base or dimensionless unit within a unit-system using [ut_new_base_unit(), page 8] or [ut_new_dimensionless_unit(), page 8]—you'll just also have to map identifiers to the newly-created unit in order to be able to retrieve it later by identifier.

`ut_unit* ut_new_base_unit (ut_system* system)` [Function]

Creates and adds a new base-unit to the unit-system referenced by *system*. This function returns the new base-unit. You should pass the returned pointer to `ut_free()` when you no longer need the unit. If an error occurs, then this function writes an error-message using [ut_handle_error_message(), page 30] and returns `NULL`. Also, [ut_get_status(), page 29] will return one of the following:

`UT_BAD_ARG`
: *system* is `NULL`.

`UT_OS`
: Operating-system failure. See `errno`.

If you use [ut_read_xml(), page 6], then you should not normally need to call this function.

`ut_unit* ut_new_dimensionless_unit (ut_system* system)` [Function]

Creates and adds a new dimensionless-unit to the unit-system referenced by *system*. This function returns the new dimensionless-unit. You should pass the returned pointer to `ut_free()` when you no longer need the unit. If an error occurs, then this function writes an error-message using [ut_handle_error_message(), page 30] and returns `NULL`. Also, [ut_get_status(), page 29] will return one of the following:

`UT_BAD_ARG`
: *system* is `NULL`.

`UT_OS`
: Operating-system failure. See `errno`.

If you use [ut_read_xml(), page 6], then you should not normally need to call this function.

3.4 Adding Unit-Prefixes to a Unit-System

A prefix is a word or symbol that is appended to the beginning of a word or symbol that represents a unit in order to modify the value of that unit. For example, the prefix "kilo" in the word "kiloamperes" changes the value from one ampere to one-thousand amperes.

If you use [ut_read_xml()], page 6, then you should not normally need to add any new prefixes to a unit-system.

[ut_status], page 29 ut_add_name_prefix (ut_system* *system*, const char* *name*, double *value*) [Function]

Adds the name-prefix *name* with the value *value* to the unit-system *system*. A name-prefix is something like "mega" or "milli". Comparisons between name-prefixes are case-insensitive. This function returns one of the following:

UT_SUCCESS
: Success.

UT_BAD_ARG
: *system* or *name* is NULL, or *value* is 0.

UT_EXISTS
: *name* already maps to a different value.

UT_OS
: Operating-system failure. See errno.

[ut_status], page 29 ut_add_symbol_prefix (ut_system* *system*, const char* *symbol*, double *value*) [Function]

Adds the symbol-prefix *symbol* with the value *value* to the unit-system *system*. A symbol-prefix is something like "M" or "m". Comparisons between symbol-prefixes are case-sensitive. This function returns one of the following:

UT_SUCCESS
: Success.

UT_BAD_ARG
: *system* or *symbol* is NULL, or *value* is 0.

UT_EXISTS
: *symbol* already maps to a different value.

UT_OS
: Operating-system failure. See errno.

3.5 Miscelaneous Operations on Unit-Systems

void ut_free_system (ut_system* *system*) [Function]

Frees the unit-system referenced by *system*. All unit-to-identifier and identifier-to-unit mappings are removed. Use of system after this function returns results in undefined behavior.

[ut_status], page 29 ut_set_second (const ut_unit* *second*) [Function]

Sets the "second" unit of a unit-system. This function must be called before the first call to ut_offset_by_time() for a unit in the same unit-system. [ut_read_xml()], page 6 calls this function if the unit-system it's reading contains a unit named "second". This function returns one of the following:

`UT_SUCCESS`
: The "second" unit of *system* was successfully set.

`UT_EXISTS`
: The "second" unit of *system* is set to a different unit.

`UT_BAD_ARG`
: *second* is `NULL`.

4 Converting Values Between Units

You can convert numeric values in one unit to equivalent values in another, compatible unit by means of a converter. For example

```
#include <udunits2.h>
...
    ut_unit*      from = ...;
    ut_unit*      to = ...;
    cv_converter* converter = ut_get_converter(from, to);
    double       fromValue = ...;
    double       toValue = cv_convert_double(converter, fromValue);

    cv_free(converter);
```

The converter API is declared in the header-file `<converter.h>`, which is automatically included by the UDUNITS-2 header-file (`<udunits2.h>`) so you don't need to explicitly include it.

`int ut_are_convertible (const ut_unit* unit1, uconst t_unit* unit2)` [Function]

Indicates if numeric values in unit *unit1* are convertible to numeric values in unit *unit2* via [ut_get_converter()], page 11. In making this determination, dimensionless units are ignored. This function returns a non-zero value if conversion is possible; otherwise, `0` is returned and [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *unit1* or *unit2* is `NULL`.

`UT_NOT_SAME_SYSTEM`
: *unit1* and *unit2* belong to different [unit-system], page 5s.

`UT_SUCCESS`
: Conversion between the units is not possible (e.g., *unit1* refers to a meter and *unit2* refers to a kilogram.

`cv_converter* ut_get_converter (ut_unit* const from, ut_unit* const to)` [Function]

Creates and returns a converter of numeric values in the *from* unit to equivalent values in the *to* unit. You should pass the returned pointer to `cv_free()` when you no longer need the converter. If an error occurs, then this function writes an error-message using [ut_handle_error_message()], page 30 and returns `NULL`. Also, [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *from* or *to* is `NULL`.

`UT_NOT_SAME_SYSTEM`
: The units *from* and *to* don't belong to the same unit-system.

`UT_MEANINGLESS`
: The units belong to the same unit-system but conversion between them is meaningless (e.g., conversion between seconds and kilograms is meaningless).

`UT_OS` Operating-system failure. See `errno`.

`float cv_convert_float (const cv_converter* converter, const float value)` [Function]

Converts the single floating-point value *value* and returns the new value.

`double cv_convert_double (const cv_converter* converter, const double value)` [Function]

Converts the single double-precision value *value* and returns the new value.

`float* cv_convert_floats (const cv_converter* converter, const float* in, size_t count, float* out)` [Function]

Converts the *count* floating-point values starting at *in*, writing the new values starting at *out* and, as a convenience, returns *out*. The input and output arrays may overlap or be identical.

`double* cv_convert_doubles (const cv_converter* converter, const double* in, size_t count, double* out)` [Function]

Converts the *count* double-precision values starting at *in*, writing the new values starting at *out* and, as a convenience, returns *out*. The input and output arrays may overlap or be identical.

`void cv_free (cv_converter* conv);` [Function]

Frees resources associated with the converter referenced by *conv*. You should call this function when you no longer need the converter. Use of *conv* upon return results in undefined behavior.

5 Parsing a String into a Unit

Here's an example of parsing a string representation of a unit into its binary representation:

```
#include <stdlib.h>
#include <udunits2.h>
...
    ut_system*   unitSystem = [ut_read_xml()], page 6;
    const char* string = "kg.m2/s3";
    ut_unit*     watt = [ut_parse()], page 13(unitSystem, string, UT_ASCII);

    if (watt == NULL) {
        /* Unable to parse string. */
    }
    else {
        /* Life is good. */
    }
```

`ut_unit* ut_parse (const ut_system* system, const char* string, ut_encoding encoding)` [Function]

Returns the binary unit representation corresponding to the string unit representation *string* in the character-set *encoding* using the unit-system *system*. *string* must have no leading or trailing whitespace (see [ut_trim()], page 13). If an error occurs, then this function returns `NULL` and [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *system* or *string* is `NULL`.

`UT_SYNTAX`
: *string* contained a syntax error.

`UT_UNKNOWN`
: *string* contained an unknown identifier.

`UT_OS` Operating-system failure. See `errno` for the reason.

`size_t ut_trim (char* string, ut_encoding encoding)` [Function]

Removes all leading and trailing whitespace from the NUL-terminated string *string*. Returns *string*, which is modified if it contained leading or trailing whitespace.

6 Unit Syntax

For the most part, the UDUNITS-2 package follows the syntax for unit-strings promulgated by the US National Institute for Standards and Technology (NIST). Details, of which, can be found at http://physics.nist.gov/cuu/Units/index.html. The one general exception to this is the invention of a syntax for "offset"-units (e.g., the definition of the degree Celsius is "K @ 273.15").

6.1 Unit Specification Examples

String Type	Using Names	Using Symbols	Comment
Simple	meter	m	
Raised	meter^2	m2	higher precedence than multiplying or dividing
Product	newton meter	N.m	
Quotient	meter per second	m/s	
Scaled	60 second	60 s	
Prefixed	kilometer	km	
Offset	kelvin from 273.15	K @ 273.15	lower precedence than multiplying or dividing
Logarithmic	lg(re milliwatt)	lg(re mW)	"lg" is base 10, "ln" is base e, and "lb" is base 2
Grouped	(5 meter)/(30 second)	(5 m)/(30 s)	

The above may be combined, e.g., "0.1 lg(re m/(5 s)^2) @ 50".

You may also look at the <def> elements in Chapter 12 [Database], page 31 to see examples of string unit specifications.

You may use the ⟨undefined⟩ [udunits2], page ⟨undefined⟩ utility to experiment with string unit specifications.

6.2 Unit Grammar

Here is the unit-syntax understood by the UDUNITS-2 package. Words printed *Thusly* indicate non-terminals; words printed THUSLY indicate terminals; and words printed <thusly> indicate lexical elements.

```
Unit-Spec: one of
        nothing
        Shift-Spec

Shift-Spec: one of
        Product-Spec
        Product-Spec SHIFT REAL
        Product-Spec SHIFT INT
        Product-Spec SHIFT Timestamp

Product-Spec: one of
        Power-Spec
        Product-Spec Power-Spec
```

```
        Product-Spec MULTIPLY Power-Spec
        Product-Spec DIVIDE Power-Spec

Power-Spec: one of
        Basic-Spec
        Basic-Spec INT
        Basic-Spec EXPONENT
        Basic-Spec RAISE INT

Basic-Spec: one of
        ID
        "(" Shift-Spec ")"
        LOGREF Product_Spec ")"
        Number

Number: one of
        INT
        REAL

Timestamp: one of
        DATE
        DATE CLOCK
        DATE CLOCK CLOCK
        DATE CLOCK INT
        DATE CLOCK ID
        TIMESTAMP
        TIMESTAMP INT
        TIMESTAMP ID

SHIFT:
        <space>* <shift_op> <space>*

<shift_op>: one of
        "@"
        "after"
        "from"
        "since"
        "ref"

REAL:
        the usual floating-point format

INT:
        the usual integer format

MULTIPLY: one of
        "-"
```

```
        "."
        "*"
        <space>+
        <centered middot>

DIVIDE:
        <space>* <divide_op> <space>*

<divide_op>: one of
        per
        PER
        "/"

EXPONENT:
        ISO-8859-9 or UTF-8 encoded exponent characters

RAISE: one of
        "^"
        "**"

ID: one of
        <id>
        "%"
        "'"
        "\""
        degree sign
        greek mu character

<id>:
        <alpha> <alphanum>*

<alpha>:
        [A-Za-z_]
        ISO-8859-1 alphabetic characters
        non-breaking space

<alphanum>: one of
        <alpha>
        <digit>

<digit>:
        [0-9]

LOGREF:
        <log> <space>* <logref>

<log>: one of
```

```
        "log"
        "lg"
        "ln"
        "lb"

<logref>:
        "(" <space>* <re> ":"? <space>*

DATE:
        <year> "-" <month> ("-" <day>)?

<year>:
        [+-]?[0-9]{1,4}

<month>:
        "0"?[1-9]|1[0-2]

<day>:
        "0"?[1-9]|[1-2][0-9]|"30"|"31"

CLOCK:
        <hour> ":" <minute> (":" <second>)?

TIMSTAMP:
        <year> (<month> <day>?)? "T" <hour> (<minute> <second>?)?

<hour>:
        [+-]?[0-1]?[0-9]|2[0-3]

<minute>:
        [0-5]?[0-9]

<second>:
        (<minute>|60) (\.[0-9]*)?
```

7 Formatting a Unit into a String

Use the [ut_format()], page 18 function to obtain the string representation of a binary unit. For example, the following gets the definition of the unit "watt" in ASCII characters using unit-symbols rather than unit-names:

```
ut_unit*     watt = ...;
char         buf[128];
unsigned     opts = [ut_encoding], page 32 | UT_DEFINITION;
int          len = [ut_format()], page 18(watt, buf, sizeof(buf), opts);

if (len == -1) {
    /* Couldn't get string */
}
else if (len == sizeof(buf)) {
    /* Entire buffer used: no terminating NUL */
}
else {
    /* Have string with terminating NUL */
}
```

`int ut_format (const ut_unit* unit, char* buf, size_t size, unsigned opts)` [Function]

Formats the unit *unit* (i.e., returns its string representation) into the buffer pointed-to by *buf* of size *size*. The argument *opts* specifies how the formatting is to be done and is a bitwise OR of a [ut_encoding], page 32 value and zero or more of the following:

`UT_NAMES` Use unit names instead of symbols.

`UT_DEFINITION`
: The formatted string should be the definition of *unit* in terms of basic-units instead of stopping any expansion at the highest level possible.

On succes, this function returns the number of characters written into *buf*, which will be less than or equal to *size*. If the number is equal to *size*, then the buffer is too small to have a terminating `NUL` character.

On failure, this function returns `-1` and [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *unit* or *buf* is `NULL`, or *opts* contains the bit patterns of both `UT_LATIN1` and `UT_UTF8`.

`UT_CANT_FORMAT`
: *unit* can't be formatted in the desired manner (e.g., *opts* contains `UT_ASCII` but *unit* doesn't have an identifier in that character-set or *opts* doesn't contain UT_NAMES and a necessary symbol doesn't exist).

8 Unit Operations

You can use unit operations to construct new units, get information about units, or compare units.

8.1 Unary Unit Operations

`void ut_free (ut_unit* unit)` [Function]

Frees resources associated with *unit*. You should invoke this function on every unit that you no longer need. Use of *unit* upon return from this function results in undefined behavior.

`ut_unit* ut_scale (double factor, const ut_unit* unit)` [Function]

Returns a unit equivalent to another unit scaled by a numeric factor. For example:

```
const ut_unit*   meter = ...
const ut_unit*   kilometer = ut_scale(1000, meter);
```

The returned unit is equivalent to *unit* multiplied by *factor*. You should pass the returned pointer to [ut_free()], page 19 when you no longer need the unit.

`ut_unit* ut_offset (const ut_unit* unit, double offset)` [Function]

Returns a unit equivalent to another unit relative to a particular origin. For example:

```
const ut_unit*   kelvin = ...
const ut_unit*   celsius = ut_offset(kelvin, 273.15);
```

The returned unit is equivalent to *unit* with an origin of *offset*. You should pass the returned pointer to [ut_free()], page 19 when you no longer need the unit. If an error occurs, then this function returns `NULL` and [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *unit* is `NULL`.

`UT_OS`
: Operating-system error. See `errno` for the reason.

`ut_unit* ut_offset_by_time (const ut_unit* const unit, const double origin)` [Function]

Returns a timestamp-unit equivalent to the time unit *unit* referenced to the time-origin *origin* (as returned by [ut_encode_time()], page 28). For example:

```
const ut_unit*   second = ...
const ut_unit*   secondsSinceTheEpoch =
    ut_offset_by_time(second, ut_encode_time(1970, 1, 1, 0, 0, 0.0));
```

Leap seconds are not taken into account. You should pass the returned pointer to [ut_free()], page 19 when you no longer need the unit. If an error occurs, then this function returns `NULL` and [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *unit* is `NULL`.

`UT_OS`
: Operating-system error. See `errno` for the reason.

`UT_MEANINGLESS`
: Creation of a timestamp unit based on *unit* is not meaningful. It might not be a time-unit, for example.

`UT_NO_SECOND`
: The associated unit-system doesn't contain a "second" unit. See [ut_set_second()], page 9.

CAUTION: The timestamp-unit was created to be analogous to, for example, the degree celsius—but for the time dimension. I've come to believe, however, that creating such a unit was a mistake, primarily because users try to use the unit in ways for which it was not designed (such as converting dates in a calendar whose year is exactly 365 days long). Such activities are much better handled by a dedicated calendar package. Please be careful about using timestamp-units.

`ut_unit* ut_invert (const ut_unit* unit)` [Function]
Returns the inverse (i.e., reciprocal) of the unit *unit*. This convenience function is equal to [ut_raise()], page 20. You should pass the returned pointer to [ut_free()], page 19 when you no longer need the unit. If an error occurs, then this function writes an error-message using [ut_handle_error_message()], page 30 and returns `NULL`. Also, [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *unit* is `NULL`.

`UT_OS` Operating-system error. See `errno` for the reason.

`ut_unit* ut_raise (const ut_unit* unit, int power)` [Function]
Returns the unit equal to unit *unit* raised to the power *power*. You should pass the returned pointer to `ut_free()` when you no longer need the unit. If an error occurs, then this function writes an error-message using [ut_handle_error_message()], page 30 and returns `NULL`. Also, [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *unit* is `NULL`.

`UT_OS` Operating-system error. See `errno` for the reason.

`ut_unit* ut_root (const ut_unit* unit, int root)` [Function]
Returns the unit equal to the *root* root of unit *unit*. You should pass the returned pointer to `ut_free()` when you no longer need the unit. If an error occurs, then this function writes an error-message using [ut_handle_error_message()], page 30 and returns `NULL`. Also, [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *unit* is `NULL`.

`UT_MEANINGLESS`
: It's meaningless to take the given root of the given unit. This could be because the resulting unit would have fractional (i.e., non-integral) dimensionality, or because the unit is, for example, a logarithmic unit.

`UT_OS` Operating-system error. See `errno` for the reason.

`ut_unit* ut_log (double` *base*`, const ut_unit*` *reference*`)` [Function]

Returns the logarithmic unit corresponding to the logarithmic base *base* and a reference level specified as the unit *reference*. For example, the following creates a decibel unit with a one milliwatt reference level:

```
const ut_unit* milliWatt = ...;
const ut_unit* bel_1_mW = ut_log(10.0, milliWatt);

if (bel_1_mW != NULL) {
    const ut_unit* decibel_1_mW = [ut_scale()], page 19(0.1, bel_1_mW);

    [ut_free()], page 19(bel_1_mW);   /* no longer needed */

    if (decibel_1_mW != NULL) {
        /* Have decibel unit with 1 mW reference */
        ...
        [ut_free()], page 19(decibel_1_mW);
    }                 /* "decibel_1_mW" allocated */
}
```

You should pass the returned pointer to `ut_free()` when you no longer need the unit. If an error occurs, then this function writes an error-message using [ut_handle_error_message()], page 30 and returns `NULL`. Also, [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *reference* is `NULL`.

`UT_OS` Operating-system error. See `errno` for the reason.

`UT_BAD_ARG`
: *base* is invalid (e.g., it must be greater than one).

`const char* ut_get_name (const ut_unit*` *unit*`, ut_encoding` *encoding*`)` [Function]

Returns the name to which the unit referenced by *unit* maps in the character-encoding specified by *encoding*. If this function returns `NULL`, then [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *name* is `NULL`.

`UT_SUCCESS`
: *unit* doesn't map to a name in the given character-set.

`const char* ut_get_symbol (const ut_unit*` *unit*`, ut_encoding` *encoding*`)` [Function]

Returns the symbol to which the unit referenced by *unit* maps in the character-encoding specified by *encoding*. If this function returns `NULL`, then [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *symbol* is `NULL`.

`UT_SUCCESS`
: *unit* doesn't map to a symbol in the given character-set.

`ut_system* ut_get_system (const ut_unit* unit)` [Function]
: Returns the unit-system to which the unit referenced by *unit* belongs. If *unit* is `NULL`, then this function writes an error-message using [ut_handle_error_message()], page 30 and returns `NULL`. Also, [ut_get_status()], page 29 will return `UT_BAD_ARG`.

`int ut_is_dimensionless (const ut_unit* unit)` [Function]
: Indicates if unit *unit* is dimensionless (like "radian"). This function returns a non-zero value if the unit is dimensionfull; otherwise, `0` is returned and [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *unit1* is `NULL`.

`UT_SUCCESS`
: The unit is dimensionless.

`ut_unit* ut_clone (const ut_unit* unit)` [Function]
: Returns a copy of the unit referenced by *unit*. You should pass the returned pointer to `ut_free()` when you no longer need the unit. If an error occurs, then this function writes an error-message using [ut_handle_error_message()], page 30 and returns `NULL`. Also, [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *unit* is `NULL`.

`UT_OS` Operating-system failure. See `errno`.

If you use [ut_read_xml()], page 6, then you should not normally need to call this function.

[ut_status], page 29 `ut_accept_visitor (const ut_unit* unit, const` [ut_visitor], page 22`* visitor, void* arg)` [Function]
: Accepts the visitor *visitor* to the unit *unit*. The argument *arg* is passed to the visitor's functions. This function returns one of the following:

`UT_BAD_ARG`
: *visitor* or *unit* is `NULL`.

`UT_VISIT_ERROR`
: An error occurred in *visitor* while visiting *unit*.

`UT_SUCCESS`
: Success.

`ut_visitor` *int foo*(*int*) *int bar*(*int*, *int*) [Data type]
: You pass a pointer to a data object of this type if and when you call [ut_accept_visitor()], page 22. It contains the following pointers to functions that implement your unit-visitor:

[ut_status], page 29 (*visit_basic)(const ut_unit* *unit*, void* *arg*);
: Visits the basic-unit *unit*. A basic-unit is a base unit like "meter" or a non-dimensional but named unit like "radian". This function returns [ut_status], page 29 on and only on success.

[ut_status], page 29 (*visit_product)(const ut_unit* *unit*, int *count*, const ut_unit* const* *basicUnits*, const int* *powers*, void* *arg*);
: Visits the product-unit *unit*. The product-unit is a product of the *count* basic-units referenced by *basicUnits*, each raised to their respective, non-zero power in *powers*. This function returns [ut_status], page 29 on and only on success.

[ut_status], page 29 (*visit_galilean)(const ut_unit* *unit*, double *scale*, const ut_unit* *underlyingUnit*, double *origin*, void* arg);
: Visits the Galilean-unit *unit*. The Galilean-unit has the underlying unit *underlyingUnit* and either the non-unity scale factor *scale* or the non-zero origin *origin*, or both. This function returns [ut_status], page 29 on and only on success.

[ut_status], page 29 (*visit_timestamp)(const ut_unit* *unit*, const ut_unit* *timeUnit*, double *origin*, void* *arg*);
: Visits the timestamp-unit *unit*. The timestamp-unit has the underlying unit of time *timeUnit* and the [ut_encode_time()], page 28-encoded time-origin *origin*. This function returns [ut_status], page 29 on and only on success.

[ut_status], page 29 (*visit_logarithmic)(const ut_unit* *unit*, double *base*, const ut_unit* *reference*, void* *arg*);
: Visits the logarithmic-unit *unit*. The logarithmic-unit has the logarithmic base *base* and the reference-level is specified by the unit *reference*. This function returns [ut_status], page 29 on and only on success.

8.2 Binary Unit Operations

Binary unit operations act on two units. The functions [ut_are_convertible()], page 11 and [ut_get_converter()], page 11 are also binary unit operations but they are documented elsewhere.

ut_unit* ut_multiply (const ut_unit* *unit1*, const ut_unit* *unit2*) [Function]
: Returns the result of multiplying unit *unit1* by unit *unit2*. You should pass the pointer to [ut_free()], page 19 when you no longer need the unit On failure, this function returns NULL and [ut_get_status()], page 29 will return one of the following:

UT_BAD_ARG
: *unit1* or *unit2* is NULL.

UT_NOT_SAME_SYSTEM
: *unit1* and *unit2* belong to different [unit-system], page 5s.

UT_OS
: Operating-system error. See *errno* for the reason.

`ut_unit* ut_divide (const ut_unit* numer, const ut_unit* denom)` [Function]

Returns the result of dividing unit *numer* by unit *denom*. You should pass the pointer to [ut_free()], page 19 when you no longer need the unit On failure, this function returns `NULL` and [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *numer* or *denom* is `NULL`.

`UT_NOT_SAME_SYSTEM`
: *unit1* and *unit2* belong to different [unit-system], page 5s.

`UT_OS` Operating-system error. See `errno` for the reason.

`int ut_compare (const ut_unit* unit1, const ut_unit* unit2)` [Function]

Compares two units. Returns a value less than, equal to, or greater than zero as *unit1* is considered less than, equal to, or greater than *unit2*, respectively. Units from different [unit-system], page 5s never compare equal. The value zero is also returned if both unit pointers are `NULL`.

`int ut_same_system (const ut_unit* unit1, const ut_unit* unit2)` [Function]

Indicates if two units belong to the same unit-system. This function returns a non-zero value if the two units belong to the same [unit-system], page 5; otherwise, `0` is returned and [ut_get_status()], page 29 will return one of the following:

`UT_BAD_ARG`
: *unit1* or *unit2* is `NULL`.

`UT_SUCCESS`
: The units belong to different [unit-system], page 5s.

9 Mapping Between Identifiers and Units

Within a unit-system, you can map an identifier to a unit and vice versa. If an identifier maps to a unit, then the unit can be retrieved from the unit-system via the identifier. Similarly, if a unit maps to an identifier, then the unit can be printed using the identifier.

There a two kinds of identifiers: names and symbols.

9.1 Names

You can map a name to a unit and vice versa. If you use [ut_read_xml()], page 6, then you shouldn't normally need to do this.

[ut_status], page 29 ut_map_name_to_unit (const char* *name*, const ut_encoding *encoding*, const ut_unit* *unit*) [Function]

Maps the name referenced by *name*, in character-set *encoding*, to the unit referenced by *unit* in the unit-system that contains *unit*. This function returns one of the following:

UT_SUCCESS
: Success.

UT_BAD_ARG
: *name* or *unit* is NULL.

UT_OS
: Operating-system failure. See errno.

UT_EXISTS
: *name* already maps to a different unit.

[ut_status], page 29 ut_unmap_name_to_unit (ut_system* *system*, const char* *name*, const ut_encoding *encoding*) [Function]

Removes any mapping from name *name*, in character-set *encoding*, to a unit in unit-system *system*. This function returns one of the following:

UT_SUCCESS
: Success.

UT_BAD_ARG
: *system* or *name* is NULL.

[ut_status], page 29 ut_map_unit_to_name (const ut_unit* *unit*, const char* *name*, ut_encoding *encoding*) [Function]

Maps the unit *unit* to the name *name*, which is in character-set *encoding*, in the unit-system that contains the unit. This function returns one of the following:

UT_SUCCESS
: Success.

UT_BAD_ARG
: *unit* or *name* is NULL, or *name* is not in the character-set *encoding*.

UT_OS
: Operating-system failure. See errno.

UT_EXISTS
: *unit* already maps to a different name.

[ut_status], page 29 ut_unmap_unit_to_name (const ut_unit* *unit*, ut_encoding *encoding*) [Function]

Removes any mapping from unit *unit* to a name in character-set *encoding* from the unit-system that contains the unit. This function returns one of the following:

UT_SUCCESS
: Success.

UT_BAD_ARG
: *unit* is NULL.

9.2 Symbols

You can map a symbol to a unit and vice versa. If you use [ut_read_xml()], page 6, then you shouldn't normally need to do this.

[ut_status], page 29 ut_map_symbol_to_unit (const char* *symbol*, const ut_encoding *encoding*, const ut_unit* *unit*) [Function]

Maps the symbol referenced by *symbol*, in character-set *encoding*, to the unit referenced by *unit* in the unit-system that contains *unit*. This function returns one of the following:

UT_SUCCESS
: Success.

UT_BAD_ARG
: *symbol* or *unit* is NULL.

UT_OS
: Operating-system failure. See errno.

UT_EXISTS
: *symbol* already maps to a different unit.

[ut_status], page 29 ut_unmap_symbol_to_unit (ut_system* *system*, const char* *symbol*, const ut_encoding *encoding*) [Function]

Removes any mapping from symbol *symbol*, in character-set *encoding*, to a unit in unit-system *system*. This function returns one of the following:

UT_SUCCESS
: Success.

UT_BAD_ARG
: *system* or *symbol* is NULL.

[ut_status], page 29 ut_map_unit_to_symbol (const ut_unit* *unit*, const char* *symbol*, ut_encoding *encoding*) [Function]

Maps the unit *unit* to the symbol *symbol*, which is in character-set *encoding*, in the unit-system that contains the unit. This function returns one of the following:

UT_SUCCESS
: Success.

UT_BAD_ARG
: *unit* or *symbol* is NULL.

`UT_BAD_ARG`
: Symbol *symbol* is not in the character-set *encoding*.

`UT_OS`
: Operating-system failure. See `errno`.

`UT_EXISTS`
: *unit* already maps to a different symbol.

`[ut_status], page 29` `ut_unmap_unit_to_symbol (const ut_unit* unit, ut_encoding encoding)` [Function]

Removes any mapping from unit *unit* to a symbol in character-set *encoding* from the unit-system that contains the unit. This function returns one of the following:

`UT_SUCCESS`
: Success.

`UT_BAD_ARG`
: *unit* is `NULL`.

10 The Handling of Time

In general, the UDUNITS-2 package handles time by encoding it as double-precision value, which can then be acted upon arithmetically.

`double ut_encode_time (int year, int month, int day, int hour, int minute, double second)` [Function]

Encodes a time as a double-precision value. This convenience function is equivalent to

```
[ut_encode_date()], page 28(year,month,day) + [ut_encode_clock()],
page 28(hour,minute,second)
```

`double ut_encode_date (int year, int month, int day)` [Function]

Encodes a date as a double-precision value. You probably won't use this function.

`double ut_encode_clock (int hour, int minute, double second)` [Function]

Encodes a clock-time as a double-precision value. You probably won't use this function.

`void ut_decode_time (double time, int* year, int* month, int* day, int* hour, int* minute, double* second, double* resolution)` [Function]

Decodes a time from a double-precision value into its individual components. The variable referenced by *resolution* will be set to the resolution (i.e., uncertainty) of the time in seconds.

11 Error Handling

Error-handling in the units module has two aspects: the status of the last operation performed by the module and the handling of error-messages:

11.1 Status of Last Operation

UDUNITS-2 functions set their status by calling [ut_set_status()], page 29. You can use the function [ut_get_status()], page 29 to retrieve that status.

[ut_status], page 29 ut_get_status (void) [Function]
Returns the value specified in the last call to [ut_set_status()], page 29

void ut_set_status ([ut_status], page 29 *status*) [Function]
Set the status of the units module to *status*.

ut_status [Data type]
This enumeration has the following values:

UT_SUCCESS
: Success

UT_BAD_ARG
: An argument violates the the function's contract (e.g., it's NULL).

UT_EXISTS
: Unit, prefix, or identifier already exists

UT_NO_UNIT
: No such unit exists

UT_OS
: Operating-system error. See errno for the reason.

UT_NOT_SAME_SYSTEM
: The units belong to different unit-systems

UT_MEANINGLESS
: The operation on the unit or units is meaningless

UT_NO_SECOND
: The unit-system doesn't have a unit named "second"

UT_VISIT_ERROR
: An error occurred while visiting a unit

UT_CANT_FORMAT
: A unit can't be formatted in the desired manner

UT_SYNTAX
: String unit representation contains syntax error

UT_UNKNOWN
: String unit representation contains unknown word

UT_OPEN_ARG
: Can't open argument-specified unit database

`UT_OPEN_ENV`
: Can't open environment-specified unit database

`UT_OPEN_DEFAULT`
: Can't open installed, default, unit database

`UT_PARSE` Error parsing unit database

11.2 Error-Messages

`int ut_handle_error_message (const char* fmt, ...)` [Function]

Handles the error-message corresponding to the format-string *fmt* and any subsequent arguments referenced by it. The interpretation of the formatting-string is identical to that of the UNIX function `printf()`. On success, this function returns the number of bytes in the error-message; otherwise, this function returns `-1`.

Use the function [ut_set_error_message_handler()], page 30 to change how error-messages are handled.

[ut_error_message_handler], page 30 [Function]
: `ut_set_error_message_handler ([ut_error_message_handler], page 30 handler)`

Sets the function that handles error-messages and returns the previous error-message handler. The initial error-message handler is [ut_write_to_stderr()], page 30.

`int ut_write_to_stderr (const char* fmt, va_list args)` [Function]

Writes the variadic error-message corresponding to formatting-string *fmt* and arguments *args* to the standard-error stream and appends a newline. The interpretation of the formatting-string is identical to that of the UNIX function `printf()`. On success, this function returns the number of bytes in the error-message; otherwise, this function returns `-1`.

`int ut_ignore (const char* fmt, va_list args)` [Function]

Does nothing. In particular, it ignores the variadic error-message corresponding to formatting-string *fmt* and arguments *args*. Pass this function to [ut_set_error_message_handler()], page 30 when you don't want the unit module to print any error-messages.

`ut_error_message_handler` [Data type]

This is the type of an error-message handler. It's definition is

```
typedef int (*ut_error_message_handler)(const char* fmt, va_list args);
```

12 The Units Database

The database of units that comes with the UDUNITS-2 package is an XML-formatted file that is based on the SI system of units. It contains the names and symbols of most of the units that you will ever encounter. The pathname of the installed file is *datadir*`/udunits2.xml`, where *datadir* is the installation-directory for read-only, architecture-independent data (e.g., `/usr/local/share`). This pathname is the default that `[ut_read_xml()], page 6` uses.

Naturally, because the database is a regular file, it can be edited to add new units or remove existing ones. Be very careful about doing this, however, because you might lose the benefit of exchanging unit-based information with others who haven't modified their database.

13 Data Types

The data types [ut_visitor], page 22, [ut_status], page 29, and [ut_error_message_handler], page 30 are documented elsewhere.

`ut_encoding` [Data type]

This enumeration has the following values:

`UT_ASCII` US ASCII character-set.

`UT_ISO_8859_1`
The ISO-8859-1 character-set.

`UT_LATIN1`
Synonym for `UT_ISO_8859_1`.

`UT_UTF8` The UTF-8 encoding of the Unicode character-set.

Index