Units Conversion

Edition 2.00 for `units` Version 2.00

Adrian Mariano

Units Conversion

1 Overview of `units`

The `units` program converts quantities expressed in various systems of measurement to their equivalents in other systems of measurement. Like many similar programs, it can handle multiplicative scale changes. It can also handle nonlinear conversions such as Fahrenheit to Celsius.[1] See Section 6.1 [Temperature Conversions], page 11. The program can also perform conversions from and to sums of units, such as converting between meters and feet plus inches.

Beyond simple unit conversions, `units` can be used as a general-purpose scientific calculator that keeps track of units in its calculations. You can form arbitrary complex mathematical expressions of dimensions including sums, products, quotients, powers, and even roots of dimensions. Thus you can ensure accuracy and dimensional consistency when working with long expressions that involve many different units that may combine in complex ways.

The units are defined in an external data file. You can use the extensive data file that comes with this program, or you can provide your own data file to suit your needs. You can also use your own data file to supplement the standard data file.

Basic operation is simple: you enter the units that you want to convert *from* and the units that you want to convert *to*. You can use the program interactively with prompts, or you can use it from the command line.

2 Interacting with `units`

To invoke units for interactive use, type *`units`* at your shell prompt. The program will print something like this:

```
Currency exchange rates from 04/23/12
2516 units, 85 prefixes, 65 nonlinear units

You have:
```

At the '`You have:`' prompt, type the quantity and units that you are converting *from*. For example, if you want to convert ten meters to feet, type *`10 meters`*. Next, `units` will print '`You want:`'. You should type the units you want to convert *to*. To convert to feet, you would type *`feet`*. If the `readline` library was compiled in then the tab key can be used to complete unit names. See Chapter 14 [Readline Support], page 28, for more information about `readline`. To quit the program press Ctrl-C or Ctrl-D under Unix. Under Windows press Ctrl-Z.

The answer will be displayed in two ways. The first line of output, which is marked with a '`*`' to indicate multiplication, gives the result of the conversion you have asked for. The second line of output, which is marked with a '`/`' to indicate division, gives the inverse of the conversion factor. If you convert 10 meters to feet, `units` will print

[1] But Fahrenheit to Celsius is linear, you insist. Not so. A transformation T is linear if $T(x+y) = T(x)+T(y)$ and this fails for $T(x) = ax+b$. This transformation is affine, but not linear.

```
        * 32.808399
        / 0.03048
```

which tells you that 10 meters equals about 32.8 feet. The second number gives the conversion in the opposite direction. In this case, it tells you that 1 foot is equal to about 0.03 dekameters since the dekameter is 10 meters. It also tells you that 1/32.8 is about 0.03.

The `units` program prints the inverse because sometimes it is a more convenient number. In the example above, for example, the inverse value is an exact conversion: a foot is exactly 0.03048 dekameters. But the number given the other direction is inexact.

If you convert grains to pounds, you will see the following:

```
You have: grains
You want: pounds
        * 0.00014285714
        / 7000
```

>From the second line of the output you can immediately see that a grain is equal to a seven thousandth of a pound. This is not so obvious from the first line of the output. If you find the output format confusing, try using the '`--verbose`' option:

```
You have: grain
You want: aeginamina
        grain = 0.00010416667 aeginamina
        grain = (1 / 9600) aeginamina
```

If you request a conversion between units that measure reciprocal dimensions, then `units` will display the conversion results with an extra note indicating that reciprocal conversion has been done:

```
You have: 6 ohms
You want: siemens
        reciprocal conversion
        * 0.16666667
        / 6
```

Reciprocal conversion can be suppressed by using the '`--strict`' option. As usual, use the '`--verbose`' option to get more comprehensible output:

```
You have: tex
You want: typp
        reciprocal conversion
        1 / tex = 496.05465 typp
        1 / tex = (1 / 0.0020159069) typp

You have: 20 mph
You want: sec/mile
        reciprocal conversion
        1 / 20 mph = 180 sec/mile
        1 / 20 mph = (1 / 0.0055555556) sec/mile
```

If you enter incompatible unit types, the `units` program will print a message indicating that the units are not conformable and it will display the reduced form for each unit:

```
You have: ergs/hour
You want: fathoms kg^2 / day
conformability error
        2.7777778e-11 kg m^2 / sec^3
        2.1166667e-05 kg^2 m / sec
```

If you only want to find the reduced form or definition of a unit, simply press ENTER at the '`You want:`' prompt. Here is an example:

```
You have: jansky
You want:
        Definition: fluxunit = 1e-26 W/m^2 Hz = 1e-26 kg / s^2
```

The output from `units` indicates that the jansky is defined to be equal to a fluxunit which in turn is defined to be a certain combination of watts, meters, and hertz. The fully reduced (and in this case somewhat more cryptic) form appears on the far right.

Some named units are treated as dimensionless in some situations. These units include the radian and steradian. These units will be treated as equal to 1 in units conversions. Power is equal to torque times angular velocity. This conversion can only be performed if the radian is dimensionless.

```
You have: (14 ft lbf) (12 radians/sec)
You want: watts
        * 227.77742
        / 0.0043902509
```

Named dimensionless units are not treated as dimensionless in other contexts. They cannot be used as exponents so for example, '`meter^radian`' is not allowed.

If you want a list of options you can type `?` at the '`You want:`' prompt. The program will display a list of named units that are conformable with the unit that you entered at the '`You have:`' prompt above. Conformable unit *combinations* will not appear on this list.

Typing ***`help`*** at either prompt displays a short help message. You can also type ***`help`*** followed by a unit name. This will invoke a pager on the units data base at the point where that unit is defined. You can read the definition and comments that may give more details or historical information about the unit. (You can generally quit out of the page by pressing '`q`'.)

Typing ***`search`*** *text* will display a list of all of the units whose names contain *text* as a substring along with their definitions. This may help in the case where you aren't sure of the right unit name.

3 Using `units` Non-Interactively

The `units` program can perform units conversions non-interactively from the command line. To do this, type the command, type the original unit expression, and type the new units you want. If a units expression contains non-alphanumeric characters, you may need to protect it from interpretation by the shell using single or double quote characters.

If you type

```
units "2 liters" quarts
```

then `units` will print

```
        * 2.1133764
        / 0.47317647
```

and then exit. The output tells you that 2 liters is about 2.1 quarts, or alternatively that a quart is about 0.47 times 2 liters.

If the conversion is successful, then `units` will return success (zero) to the calling environment. If you enter non-conformable units then `units` will print a message giving the reduced form of each unit and it will return failure (nonzero) to the calling environment.

When you invoke `units` with only one argument, it will print out the definition of the specified unit. It will return failure if the unit is not defined and success if the unit is defined.

4 Unit Definitions

The conversion information is read from a units data file that is called '`definitions.units`' and is usually located in the '`/usr/share/units`' directory. If you invoke `units` with the '`-V`' option, it will print the location of this file. The default file includes definitions for all familiar units, abbreviations and metric prefixes. It also includes many obscure or archaic units.

Many constants of nature are defined, including these:

`pi`	ratio of circumference to diameter
`c`	speed of light
`e`	charge on an electron
`force`	acceleration of gravity
`mole`	Avogadro's number
`water`	pressure per unit height of water
`Hg`	pressure per unit height of mercury
`au`	astronomical unit
`k`	Boltzman's constant
`mu0`	permeability of vacuum
`epsilon0`	permittivity of vacuum
`G`	Gravitational constant
`mach`	speed of sound

The standard data file includes atomic masses for all of the elements and numerous other constants. Also included are the densities of various ingredients used in baking so that '`2 cups flour_sifted`' can be converted to '`grams`'. This is not an exhaustive list. Consult the units data file to see the complete list, or to see the definitions that are used.

The '`pound`' is a unit of mass. To get force, multiply by the force conversion unit '`force`' or use the shorthand '`lbf`'. (Note that '`g`' is already taken as the standard abbreviation for the gram.) The unit '`ounce`' is also a unit of mass. The fluid ounce is '`fluidounce`' or '`floz`'. British capacity units that differ from their US counterparts, such as the British Imperial gallon, are prefixed with '`br`'. Currency is prefixed with its country name: '`belgiumfranc`', '`britainpound`'.

When searching for a unit, if the specified string does not appear exactly as a unit name, then the `units` program will try to remove a trailing '`s`', '`es`'. Next units will

replace a trailing '`ies`' with '`y`'. If that fails, `units` will check for a prefix. The database includes all of the standard metric prefixes. Only one prefix is permitted per unit, so '`micromicrofarad`' will fail. However, prefixes can appear alone with no unit following them, so '`micro*microfarad`' will work, as will '`micro microfarad`'.

To find out which units and prefixes are available, read the standard units data file, which is extensively annotated.

4.1 English Customary Units

English customary units differ in various ways in different regions. In Britain a complex system of volume measurements featured different gallons for different materials such as a wine gallon and ale gallon that different by twenty percent. This complexity was swept away in 1824 by a reform that created an entirely new gallon, the British Imperial gallon defined as the volume occupied by ten pounds of water. Meanwhile in the USA the gallon is derived from the 1707 Winchester wine gallon, which is 231 cubic inches. These gallons differ by about twenty percent. By default if `units` runs in the '`en_GB`' locale you will get the British volume measures. If it runs in the '`en_US`' locale you will get the US volume measures. In other locales the default values are the US definitions. If you wish to force different definitions then set the environment variable `UNITS_ENGLISH` to either '`US`' or '`GB`' to set the desired definitions independent of the locale.

Before 1959, the value of a yard (and other units of measure defined in terms of it) differed slightly among English-speaking countries. In 1959, Australia, Canada, New Zealand, the United Kingdom, the United States, and South Africa adopted the Canadian value of 1 yard = 0.9144 m (exactly), which was approximately halfway between the values used by the UK and the US; it had the additional advantage of making 1 inch = 2.54 cm (exactly). This new standard was termed the *International Yard*. Australia, Canada, and the UK then defined all customary lengths in terms of the International Yard (Australia did not define the furlong or rod); because many US land surveys were in terms of the pre-1959 units, the US continued to define customary surveyors' units (furlong, chain, rod, and link) in terms of the previous value for the foot, which was termed the *US survey foot*. The US defined a *US survey mile* as 5280 US survey feet, and defined a *statute mile* as a US survey mile. The US values for these units differ from the international values by about 2 ppm.

The `units` program uses the international values for these units; the US values can be obtained by using either the '`US`' or the '`survey`' prefix. In either case, the simple familiar relationships among the units are maintained, e.g., 1 '`furlong`' = 660 '`ft`', and 1 '`USfurlong`' = 660 '`USft`', though the metric equivalents differ slightly between the two cases. The '`US`' prefix or the '`survey`' prefix can also be used to obtain the US survey mile and the value of the US yard prior to 1959, e.g., '`USmile`' or '`surveymile`' (but *not* '`USsurveymile`'). To get the US value of the statute mile, use either '`USstatutemile`' or '`USmile`'.

Except for distances that extend over hundreds of miles (such as in the US State Plane Coordinate System), the differences in the miles are usually insignificant:

```
You have: 100 surveymile - 100 mile
You want: inch
        * 12.672025
        / 0.078913984
```

The pre-1959 UK values for these units can be obtained with the prefix '`UK`'.

In the US, the acre is officially defined in terms of the US survey foot, but `units` uses a definition based on the international foot. If you want the official US acre use '`USacre`' and similarly use '`USacrefoot`' for the official US version of that unit. The difference between these units is about 4 parts per million.

5 Unit Expressions

5.1 Operators

You can enter more complicated units by combining units with operations such as powers, multiplication, division, addition, subtraction, and parentheses for grouping. You can use the customary symbols for these operators when `units` is invoked with its default options. Additionally, `units` supports some extensions, including high priority multiplication using a space, and a high priority numerical division operator ('`|`') that can simplify some expressions.

Powers of units can be specified using the '`^`' character as shown in the following example, or by simple concatenation of a unit and its exponent: '`cm3`' is equivalent to '`cm^3`'; if the exponent is more than one digit, the '`^`' is required. An exponent like '`2^3^2`' is evaluated right to left as usual. The '`^`' operator has the second highest precedence. You can also use '`**`' as an exponent operator.

```
You have: cm^3
You want: gallons
        * 0.00026417205
        / 3785.4118

You have: arabicfoot * arabictradepound * force
You want: ft lbf
        * 0.7296
        / 1.370614
```

You multiply units using a space or an asterisk ('`*`'). The example above shows both forms. You can divide units using the slash ('`/`') or with '`per`'.

```
You have: furlongs per fortnight
You want: m/s
        * 0.00016630986
        / 6012.8727
```

When a unit includes a prefix, exponent operators apply to the combination, so '`centimeter^3`' gives cubic centimeters. If you separate the prefix from the unit with any multiplication operator, such as '`centi meter^3`', then the prefix is treated as a separate unit, so the exponent does not apply. The second example would be a hundredth of a cubic meter, not a centimeter.

Multiplication using a space has a higher precedence than division using a slash and is evaluated left to right; in effect, the first '`/`' character marks the beginning of the denominator of a unit expression. This makes it simple to enter a quotient with several terms in

the denominator: '`W / m^2 Hz`'. If you multiply with '`*`' then you must group the terms in the denominator with parentheses: '`W / (m^2 * Hz)`'.

The higher precedence of the space operator may not always be advantageous. For example, '`m/s s/day`' is equivalent to '`m / s s day`' and has dimensions of length per time cubed. Similarly, '`1/2 meter`' refers to a unit of reciprocal length equivalent to 0.5/meter, perhaps not what you would intend if you entered that expression. The '`*`' operator is convenient for multiplying a sequence of quotients. With the '`*`' operator, the example above becomes '`m/s * s/day`', which is equivalent to '`m/day`'. Similarly, you could write '`1/2 * meter`' to get half a meter. Alternatively, parentheses can be used for grouping: you could write '`(1/2) meter`' to get half a meter. See Section 5.5 [Complicated Unit Expressions], page 9, for an illustration of the various options.

The `units` program supports another option for numerical fractions. You can indicate division of *numbers* with the vertical bar ('`|`'), so if you wanted half a meter you could write '`1|2 meter`'. This operator has the highest precedence, so you can write the square root of two thirds '`2|3^1|2`'. You cannot use the vertical bar to indicate division of non-numerical units (e.g., '`m|s`' results in an error message).

```
You have: 1|2 inch
You want: cm
        * 1.27
        / 0.78740157
```

You can use parentheses for grouping:

```
You have: (1/2) kg / (kg/meter)
You want: league
        * 0.00010356166
        / 9656.0833
```

5.2 Sums and Differences of Units

Outside of the SI, it is sometimes desirable to add values of different units. You may also wish to use `units` as a calculator that keeps track of units. Sums of conformable units are written with the '`+`' character, and differences with the '`-`' character.

```
You have: 2 hours + 23 minutes + 32 seconds
You want: seconds
        * 8612
        / 0.00011611705

You have: 12 ft + 3 in
You want: cm
        * 373.38
        / 0.0026782366

You have: 2 btu + 450 ft lbf
You want: btu
        * 2.5782804
        / 0.38785542
```

The expressions that are added or subtracted must reduce to identical expressions in primitive units, or an error message will be displayed:

```
You have: 12 printerspoint - 4 heredium
                                    ^
Illegal sum of non-conformable units
```

As usual, the precedence for '`+`' and '`-`' is lower than that of the other operators. A fractional quantity such as 2 1/2 cups can be given as '`(2+1|2) cups`'; the parentheses are necessary because multiplication has higher precedence than addition. If you omit the parentheses, `units` attempts to add '`2`' and '`1|2 cups`', and you get an error message:

```
You have: 2+1|2 cups
                  ^
Illegal sum or difference of non-conformable units
```

The expression could also be correctly written as '`(2+1/2) cups`'. If you write '`2 1|2 cups`' the space is interpreted as *multiplication* so the result is the same as '`1 cup`'.

The '`+`' and '`-`' characters sometimes appears in exponents like '`3.43e+8`'. This leads to an ambiguity in an expression like '`3e+2 yC`'. The unit '`e`' is a small unit of charge, so this can be regarded as equivalent to '`(3e+2) yC`' or '`(3 e)+(2 yC)`'. This ambiguity is resolved by always interpreting '`+`' and '`-`' as part of an exponent if possible.

5.3 Numbers as Units

For `units`, numbers are just another kind of unit. They can appear as many times as you like and in any order in a unit expression. For example, to find the volume of a box that is 2 ft by 3 ft by 12 ft in steres, you could do the following:

```
You have: 2 ft 3 ft 12 ft
You want: stere
        * 2.038813
        / 0.49048148

You have: $ 5 / yard
You want: cents / inch
        * 13.888889
        / 0.072
```

And the second example shows how the dollar sign in the units conversion can precede the five. Be careful: `units` will interpret '`$5`' with no space as equivalent to '`dollar^5`'.

5.4 Built-in Functions

Several built-in functions are provided: '`sin`', '`cos`', '`tan`', '`ln`', '`log`', '`log2`', '`exp`', '`acos`', '`atan`' and '`asin`'. The '`sin`', '`cos`', and '`tan`' functions require either a dimensionless argument or an argument with dimensions of angle.

```
You have: sin(30 degrees)
You want:
        Definition: 0.5

You have: sin(pi/2)
You want:
        Definition: 1

You have: sin(3 kg)
                  ^
Unit not dimensionless
```

The other functions on the list require dimensionless arguments. The inverse trigonometric functions return arguments with dimensions of angle.

If you wish to take roots of units, you may use the '`sqrt`' or '`cuberoot`' functions. These functions require that the argument have the appropriate root. You can obtain higher roots by using fractional exponents:

```
You have: sqrt(acre)
You want: feet
        * 208.71074
        / 0.0047913202

You have: (400 W/m^2 / stefanboltzmann)^(1/4)
You have:
        Definition: 289.80882 K

You have: cuberoot(hectare)
                           ^
Unit not a root
```

5.5 Complicated Unit Expressions

The `units` program is especially helpful in ensuring accuracy and dimensional consistency when converting lengthy unit expressions. For example, one form of the Darcy–Weisbach fluid-flow equation is

$$\Delta P = \frac{8}{\pi^2}\rho f L \frac{Q^2}{d^5}$$

where ΔP is the pressure drop, ρ is the mass density, f is the (dimensionless) friction factor, L is the length of the pipe, Q is the volumetric flow rate, and d is the pipe diameter. It might be desired to have the equation in the form

$$\Delta P = A_1 \rho f L \frac{Q^2}{d^5}$$

that accepted the user's normal units; for typical units used in the US, the required conversion could be something like

```
You have: (8/pi^2)(lbm/ft^3)ft(ft^3/s)^2(1/in^5)
You want: psi
        * 43.533969
        / 0.022970568
```

The parentheses allow individual terms in the expression to be entered naturally, as they might be read from the formula. Alternatively, the multiplication could be done with the '`*`' rather than a space; then parentheses are needed only around '`ft^3/s`' because of its exponent:

```
You have: 8/pi^2 * lbm/ft^3 * ft * (ft^3/s)^2 /in^5
You want: psi
        * 43.533969
        / 0.022970568
```

Without parentheses, and using spaces for multiplication, the previous conversion would need to be entered as

```
You have: 8 lb ft ft^3 ft^3 / pi^2 ft^3 s^2 in^5
You want: psi
        * 43.533969
        / 0.022970568
```

5.6 Backwards Compatibility: '*' and '-'

The original `units` assigned multiplication a higher precedence than division using the slash. This differs from the usual precedence rules, which give multiplication and division equal precedence, and can be confusing for people who think of units as a calculator.

The star operator ('`*`') included in this `units` program has, by default, the same precedence as division, and hence follows the usual precedence rules. For backwards compatibility you can invoke `units` with the '`--oldstar`' option. Then '`*`' has a higher precedence than division, and the same precedence as multiplication using the space.

Historically, the hyphen ('`-`') has been used in technical publications to indicate products of units, and the original `units` program treated it as a multiplication operator. Because `units` provides several other ways to obtain unit products, and because '`-`' is a subtraction operator in general algebraic expressions, `units` treats the binary '`-`' as a subtraction operator by default. For backwards compatibility use the '`--product`' option, which causes `units` to treat the binary '`-`' operator as a product operator. When '`-`' is a multiplication operator it has the same precedence as multiplication with a space, giving it a higher precedence than division.

When '`-`' is used as a unary operator it negates its operand. Regardless of the `units` options, if '`-`' appears after '`(`' or after '`+`' then it will act as a negation operator. So you can always compute 20 degrees minus 12 minutes by entering '`20 degrees + -12 arcmin`'. You must use this construction when you define new units because you cannot know what options will be in force when your definition is processed.

6 Nonlinear Unit Conversions

Nonlinear units are represented using functional notation. They make possible nonlinear unit conversions such as temperature.

6.1 Temperature Conversions

Conversions between temperatures are different from linear conversions between temperature *increments*—see the example below. The absolute temperature conversions are handled by units starting with '`temp`', and you must use functional notation. The temperature-increment conversions are done using units starting with '`deg`' and they do not require functional notation.

```
You have: tempF(45)
You want: tempC
        7.2222222

You have: 45 degF
You want: degC
        * 25
        / 0.04
```

Think of '`tempF(x)`' not as a function but as a notation that indicates that x should have units of '`tempF`' attached to it. See Section 9.3 [Defining Nonlinear Units], page 21. The first conversion shows that if it's 45 degrees Fahrenheit outside, it's 7.2 degrees Celsius. The second conversion indicates that a change of 45 degrees Fahrenheit corresponds to a change of 25 degrees Celsius. The conversion from '`tempF(x)`' is to absolute temperature, so that

```
You have: tempF(45)
You want: degR
        * 504.67
        / 0.0019814929
```

gives the same result as

```
You have: tempF(45)
You want: tempR
        * 504.67
        / 0.0019814929
```

But if you convert '`tempF(x)`' to '`degC`', the output is probably not what you expect:

```
You have: tempF(45)
You want: degC
        * 280.37222
        / 0.0035666871
```

The result is the temperature in K, because '`degC`' is defined as '`K`', the Kelvin. For consistent results, use the '`tempX`' units when converting to a temperature rather than converting a temperature increment.

6.2 Other Nonlinear Units

Some other examples of nonlinear units are numerous different ring sizes and wire gauges, the grit sizes used for abrasives, the decibel scale, shoe size, scales for the density of sugar (e.g. baume). The standard data file also supplies units for computing the area of a circle and the volume of a sphere. See the standard units data file for more details. Wire gauges with multiple zeroes are signified using negative numbers where two zeroes is '`-1`'.

Alternatively, you can use the synonyms ‘g00’, ‘g000’, and so on that are defined in the standard units data file.

```
You have: wiregauge(11)
You want: inches
        * 0.090742002
        / 11.020255

You have: brwiregauge(g00)
You want: inches
        * 0.348
        / 2.8735632

You have: 1 mm
You want: wiregauge
        18.201919

You have: grit_P(600)
You want: grit_ansicoated
        342.76923
```

The last example shows the conversion from P graded sand paper, which is the European standard and may be marked “P600” on the back, to the USA standard.

You can compute the area of a circle using the nonlinear unit, ‘circlearea’. You can also do this using the circularinch or circleinch. The next example shows two ways to compute the area of a circle with a five inch radius and one way to compute the volume of a sphere with a radius of one meter.

```
You have: circlearea(5 in)
You want: in2
        * 78.539816
        / 0.012732395

You have: 10^2 circleinch
You want: in2
        * 78.539816
        / 0.012732395

You have: spherevol(meter)
You want: ft3
        * 147.92573
        / 0.0067601492
```

7 Unit Lists: Conversion to Sums of Units

Outside of the SI, it is sometimes desirable to convert a single unit to a sum of units—for example, feet to feet plus inches. The conversion *from* sums of units was described in Section 5.2 [Sums and Differences of Units], page 7, and is a simple matter of adding the units with the ‘+’ sign:

```
You have: 12 ft + 3 in + 3|8 in
You want: ft
        * 12.28125
        / 0.081424936
```

Although you can similarly write a sum of units to convert *to*, the result will not be the conversion to the units in the sum, but rather the conversion to the particular sum that you have entered:

```
You have: 12.28125 ft
You want: ft + in + 1|8 in
        * 11.228571
        / 0.089058524
```

The unit expression given at the '`You want:`' prompt is equivalent to asking for conversion to multiples of '`1 ft + 1 in + 1|8 in`', which is 1.09375 ft, so the conversion in the previous example is equivalent to

```
You have: 12.28125 ft
You want: 1.09375 ft
        * 11.228571
        / 0.089058524
```

In converting to a sum of units like miles, feet and inches, you typically want the largest integral value for the first unit, followed by the largest integral value for the next, and the remainder converted to the last unit. You can do this conversion easily with `units` using a special syntax for lists of units. You must list the desired units in order from largest to smallest, separated by the semicolon ('`;`') character:

```
You have: 12.28125 ft
You want: ft;in;1|8 in
        12 ft + 3 in + 3|8 in
```

The conversion always gives integer coefficients on the units in the list, except possibly the last unit when the conversion is not exact:

```
You have: 12.28126 ft
You want: ft;in;1|8 in
        12 ft + 3 in + 3.00096 * 1|8 in
```

The order in which you list the units is important:

```
You have: 3 kg
You want: oz;lb
        105 oz + 0.051367866 lb

You have: 3 kg
You want: lb;oz
        6 lb + 9.8218858 oz
```

Listing ounces before pounds produces a technically correct result, but not a very useful one. You must list the units in descending order of size in order to get the most useful result.

Ending a unit list with the separator '`;`' has the same effect as repeating the last unit on the list, so '`ft;in;1|8 in;`' is equivalent to '`ft;in;1|8 in;1|8 in`'. With the example above, this gives

```
You have: 12.28126 ft
You want: ft;in;1|8 in;
        12 ft + 3 in + 3|8 in + 0.00096 * 1|8 in
```

in effect separating the integer and fractional parts of the coefficient for the last unit. If you instead prefer to round the last coefficient to an integer you can do this with the '`--round`' ('`-r`') option. With the previous example, the result is

```
You have: 12.28126 ft
You want: ft;in;1|8 in
        12 ft + 3 in + 3|8 in (rounded down to nearest 1|8 in)
```

When you use the '`-r`' option, repeating the last unit on the list has no effect (e.g., '`ft;in;1|8 in;1|8 in`' is equivalent to '`ft;in;1|8 in`'), and hence neither does ending a list with a '`;`'. With a single unit and the '`-r`' option, a terminal '`;`' *does* have an effect: it causes `units` to treat the single unit as a list and produce a rounded value for the single unit. Without the extra '`;`', the '`-r`' option has no effect on single unit conversions. This example shows the ouput using the '`-r`' option:

```
You have: 12.28126 ft
You want: in
        * 147.37512
        / 0.0067854058

You have: 12.28126 ft
You want: in;
        147 in (rounded down to nearest in)
```

Each unit that appears in the list must be conformable with the first unit on the list, and of course the listed units must also be comformable with the *You have* unit that you enter.

```
You have: meter
You want: ft;kg
             ^
conformability error
        ft = 0.3048 m
        kg = 1 kg

You have: meter
You want: lb;oz
conformability error
        1 m
        0.45359237 kg
```

In the first case, `units` reports the disagreement between units appearing on the list. In the second case, `units` reports disagreement between the unit you entered and the desired conversion. This conformability error is based on the first unit on the unit list.

Other common candidates for conversion to sums of units are angles and time:

```
You have: 23.437754 deg
You want; deg;arcmin;arcsec
    23 deg + 26 arcmin + 15.9144 arcsec
```

```
You have: 7.2319 hr
You want: hr;min;sec
    7 hr + 13 min + 54.84 sec
```

In North America, recipes for cooking typically measure ingredients by volume, and use units that are not always convenient multiples of each other. Suppose that you have a recipe for 6 and you wish to make a portion for 1. If the recipe calls for 2 1/2 cups of an ingredient, you might wish to know the measurements in terms of measuring devices you have available, you could use `units` and enter

```
You have: (2+1|2) cup / 6
You want: cup;1|2 cup;1|3 cup;1|4 cup;tbsp;tsp;1|2 tsp;1|4 tsp
        1|3 cup + 1 tbsp + 1 tsp
```

By default, if a unit in a list begins with fraction of the form $1|x$ and its multiplier is an integer, the fraction is given as the product of the multiplier and the numerator; for example,

```
You have: 12.28125 ft
You want: ft;in;1|8 in;
        12 ft + 3 in + 3|8 in
```

In many cases, such as the example above, this is what is wanted, but sometimes it is not. For example, a cooking recipe for 6 might call for 5 1/4 cup of an ingredient, but you want a portion for 2, and your 1-cup measure is not available; you might try

```
You have: (5+1|4) cup / 3
You want: 1|2 cup;1|3 cup;1|4 cup
        3|2 cup + 1|4 cup
```

This result might be fine for a baker who has a 1 1/2-cup measure (and recognizes the equivalence), but it may not be as useful to someone with more limited set of measures, who does want to do additional calculations, and only wants to know "How many 1/2-cup measures to I need to add?" After all, that's what was actually asked. With the '`--show-factor`' option, the factor will not be combined with a unity numerator, so that you get

```
You have: (5+1|4) cup / 3
You want: 1|2 cup;1|3 cup;1|4 cup
        3 * 1|2 cup + 1|4 cup
```

A user-specified fractional unit with a numerator other than 1 is never overridden, however—if a unit list specifies '`3|4 cup;1|2 cup`', a result equivalent to 1 1/2 cups will always be shown as '`2 * 3|4 cup`' whether or not the '`--show-factor`' option is given.

Some applications for unit lists may be less obvious. Suppose that you have a postal scale and wish to ensure that it's accurate at 1 oz, but have only metric calibration weights. You might try

```
You have: 1 oz
You want: 100 g;50 g; 20 g;10 g;5 g;2 g;1 g;
        20 g + 5 g + 2 g + 1 g + 0.34952312 * 1 g
```

You might then place one each of the 20 g, 5 g, 2 g, and 1 g weights on the scale and hope that it indicates close to

```
You have: 20 g + 5 g + 2 g + 1 g
You want: oz;
        0.98767093 oz
```

Appending '`;`' to '`oz`' forces a one-line display that includes the unit; here the integer part of the result is zero, so it is not displayed.

A unit list such as

```
cup;1|2 cup;1|3 cup;1|4 cup;tbsp;tsp;1|2 tsp;1|4 tsp
```

can be tedious to enter. The `units` program provides shorthand names for some common combinations:

`hms`	hours, minutes, seconds
`dms`	angle: degrees, minutes, seconds
`time`	years, days, hours, minutes and seconds
`usvol`	US cooking volume: cups and smaller

Using these shorthands, or *unit list aliases*, you can do the following conversions:

```
You have: anomalisticyear
You want: time
        1 year + 25 min + 3.4653216 sec
You have: 1|6 cup
You want: usvol
        2 tbsp + 2 tsp
```

You cannot combine a unit list alias with other units: it must appear alone at the '`You want:`' prompt.

You can display the definition of a unit list by pressing ENTER at the '`You have:`' prompt:

```
You have: dms
        Definition: unit list, deg;arcmin;arcsec
```

When you specify compact output with '`--compact`', '`--terse`' or '`-t`' and perform conversion to a unit list, `units` lists the conversion factors for each unit in the list, separated by semicolons.

```
You have: year
You want: day;min;sec
365;348;45.974678
```

Unlike the case of regular output, zeros *are* included in this output list:

```
You have: liter
You want: cup;1|2 cup;1|4 cup;tbsp
4;0;0;3.6280454
```

8 Invoking `units`

You invoke `units` like this:

```
units [options] [from-unit [to-unit]]
```

If the *from-unit* and *to-unit* are omitted, then the program will use interactive prompts to determine which conversions to perform. See Chapter 2 [Interactive Use], page 1. If

both *from-unit* and *to-unit* are given, `units` will print the result of that single conversion and then exit. If only *from-unit* appears on the command line, `units` will display the definition of that unit and exit. Units specified on the command line may need to be quoted to protect them from shell interpretation and to group them into two arguments. See Chapter 3 [Command Line Use], page 3.

The following options allow you to read in an alternative units file, check your units file, or change the output format:

`-c`
`--check` Check that all units and prefixes defined in the units data file reduce to primitive units. Print a list of all units that cannot be reduced. Also display some other diagnostics about suspicious definitions in the units data file. Only definitions active in the current locale are checked. You should always run `units` with this option after modifying a units data file.

`--check-verbose`
Like the '`--check`' option, this option prints a list of units that cannot be reduced. But to help find unit definitions that cause endless loops, it lists the units as they are checked. If `units` hangs, then the last unit to be printed has a bad definition. Only definitions active in the current locale are checked.

`-o` *format*
`--output-format` *format*
Use the specified *format* for numeric output; the format is a subset of that for the printf function in the ANSI C standard. Only a numeric format ('`E`' or '`e`' for scientific notation, '`f`' for fixed-point decimal, or '`G`' or '`g`' to specify the number of significant figures) is allowed. The default format is '`%.8g`'; for greater precision, you could specify '`-o %.15g`'. See Chapter 10 [Numeric Output Format], page 24, and the documentation for printf() for more detailed descriptions of the format specification.

`-e`
`--exponential`
Set the numeric output format to exponential (i.e., scientific notation), like that used in the Unix `units` program.

`-f` *filename*
`--file` *filename*
Instruct `units` to load the units file `filename`. You can specify up to 25 units files on the command line. When you use this option, `units` will load *only* the files you list on the command line; it will not load the standard file or your personal units file unless you explicitly list them. If *filename* is the empty string ('`-f ""`'), the default units file (or that specified by `UNITSFILE`) will be loaded in addition to any others specified with '`-f`'.

`-h`
`--help` Print out a summary of the options for `units`.

`-m`
`--minus` Causes '`-`' to be interpreted as a subtraction operator. This is the default behavior.

`-p`
`--product`
: Causes '`-`' to be interpreted as a multiplication operator when it has two operands. It will act as a negation operator when it has only one operand: '`(-3)`'. By default '`-`' is treated as a subtraction operator.

`--oldstar`
: Causes '`*`' to have the old-style precedence, higher than the precedence of division so that '`1/2*3`' will equal '`1/6`'.

`--newstar`
: Forces '`*`' to have the new (default) precedence that follows the usual rules of algebra: the precedence of '`*`' is the same as the precedence of '`/`', so that '`1/2*3`' will equal '`3/2`'.

`--compact`
: Give compact output featuring only the conversion factor. This turns off the '`--verbose`' option.

`-q`
`--quiet`
`--silent`
: Suppress prompting of the user for units and the display of statistics about the number of units loaded.

`-n`
`--nolists`
: Disable conversion to unit lists.

`-r`
`--round`
: When converting to a combination of units given by a unit list, round the value of the last unit in the list to the nearest integer.

`-S`
`--show-factor`
: When converting to a combination of units specified in a list, always show a non-unity factor before a unit that begins with a fraction with a unity denominator. By default, if the unit in a list begins with fraction of the form $1|x$ and its multiplier is an integer other than 1, the fraction is given as the product of the multiplier and the numerator (e.g., '`3|8 in`' rather than '`3 * 1|8 in`'). In some cases, this is not what is wanted; for example, the results for a cooking recipe might show '`3 * 1|2 cup`' as '`3|2 cup`'. With the '`--show-factor`' option, a result equivalent to 1.5 cups will display as '`3 * 1|2 cup`' rather than '`3|2 cup`'. A user-specified fractional unit with a numerator other than 1 is never overridden, however—if a unit list specifies '`3|4 cup;1|2 cup`', a result equivalent to 1 1/2 cups will always be shown as '`2 * 3|4 cup`' whether or not the '`--show-factor`' option is given.

`-s`
`--strict`
: Suppress conversion of units to their reciprocal units. For example, `units` will normally convert hertz to seconds because these units are reciprocals of each other. The strict option requires that units be strictly conformable to perform a conversion, and will give an error if you attempt to convert hertz to seconds.

`-1`
`--one-line`
: Give only one line of output (the forward conversion). Do not print the reverse conversion. If a reciprocal conversion is performed then `units` will still print the "reciprocal conversion" line.

`-t`
`--terse`
: Give terse output when converting units. This option can be used when calling `units` from another program so that the output is easy to parse. This option has the combined effect of these options: '`--strict`' '`--quiet`' '`--one-line`' '`--compact`'.

`-v`
`--verbose`
: Give slightly more verbose output when converting units. When combined with the '`-c`' option this gives the same effect as '`--check-verbose`'.

`-V`
`--version`
: Print program version number, tell whether the `readline` library has been included, and give the location of the default units data file.

`-l` *locale*
`--locale` *locale*
: Force a specified locale such as '`en_GB`' to get British definitions by default. This overrides the locale determined from system settings or environment variables. See Section 11.1 [Locale], page 24, for a description of locale format.

9 Adding Your Own Definitions

9.1 Units Data Files

The units and prefixes that `units` can convert are defined in the units data file, typically '`/usr/share/units/definitions.units`'. Although you can extend or modify this data file if you have appropriate user privileges, it's usually better to put extensions in separate files so that the definitions will be preserved when you update `units`.

You can include additional data files in the units database using the '`!include`' command in the standard units data file. For example

```
!include    /usr/local/share/units/local.units
```

might be appropriate for a site-wide supplemental data file. The location of the '`!include`' statement in the standard units data file is important; later definitions replace earlier ones, so any definitions in an included file will override definitions before the '`!include`' statement in the standard units data file. With normal invocation, no warning is given about redefinitions; to ensure that you don't have an unintended redefinition, run '`units -c`' after making changes to any units data file.

If you want to add your own units in addition to or in place of standard or site-wide supplemental units data files, you can include them in the '`.units`' file in your home directory.

If this file exists it is read after the standard units data file, so that any definitions in this file will replace definitions of the same units in the standard data file or in files included from the standard data file. This file will not be read if any units files are specified on the command line. (Under Windows the personal units file is named '`unitdef.units`'.)

The `units` program first tries to determine your home directory from the `HOME` environment variable. On systems running Microsoft Windows, if `HOME` does not exist, `units` attempts to find your home directory from `HOMEDRIVE` and `HOMEPATH`. Running `units -V` will display the location and name of your personal units file.

You can specify an arbitrary file as your personal units data file with the `MYUNITSFILE` environment variable; if this variable exists, its value is used without searching your home directory.

9.2 Defining New Units and Prefixes

A unit is specified on a single line by giving its name and an equivalence. Comments start with a '`#`' character, which can appear anywhere in a line. The backslash character ('\') acts as a continuation character if it appears as the last character on a line, making it possible to spread definitions out over several lines if desired. A file can be included by giving the command '`!include`' followed by the file's name. The '`!`' must be the first character on the line. The file will be sought in the same directory as the parent file unless you give a full path. The name of the file to be included cannot contain the comment character '`#`'.

Unit names must not contain any of the operator characters '`+`', '`-`', '`*`', '`/`', '`|`', '`^`', '`;`', '`~`', the comment character '`#`', or parentheses. They cannot begin or end with an underscore ('`_`'), a comma ('`,`') or a decimal point ('`.`'). Names cannot begin with a digit, and if a name ends in a digit other than zero, the digit must be preceded by a string beginning with an underscore, and afterwards consisting only of digits, decimal points, or commas. For example, '`foo_2`', '`foo_2,1`', or '`foo_3.14`' would be valid names but '`foo2`' or '`foo_a2`' would be invalid. You could define nitrous oxide as

```
N2O     nitrogen 2  + oxygen
```

but would need to define nitrogen dioxide as

```
NO_2    nitrogen + oxygen 2
```

Be careful to define new units in terms of old ones so that a reduction leads to the primitive units, which are marked with '`!`' characters. Dimensionless units are indicated by using the string '`!dimensionless`' for the unit definition.

When adding new units, be sure to use the '`-c`' option to check that the new units reduce properly. If you create a loop in the units definitions, then `units` will hang when invoked with the '`-c`' option. You will need to use the '`--check-verbose`' option, which prints out each unit as it is checked. The program will still hang, but the last unit printed will be the unit that caused the infinite loop.

If you define any units that contain '`+`' characters, carefully check them because the '`-c`' option will not catch non-conformable sums. Be careful with the '`-`' operator as well. When used as a binary operator, the '`-`' character can perform addition or multiplication depending on the options used to invoke `units`. To ensure consistent behavior use '`-`' only as a unary negation operator when writing units definitions. To multiply two units leave a space or use the '`*`' operator with care, recalling that it has two possible precedence values

and may require parentheses to ensure consistent behavior. To compute the difference of '`foo`' and '`bar`' write '`foo+(-bar)`' or even '`foo+-bar`'.

Here is an example of a short data file that defines some basic units:

```
m       !               # The meter is a primitive unit
sec     !               # The second is a primitive unit
rad     !dimensionless  # A dimensionless primitive unit
micro-  1e-6            # Define a prefix
minute  60 sec          # A minute is 60 seconds
hour    60 min          # An hour is 60 minutes
inch    0.0254 m        # Inch defined in terms of meters
ft      12 inches       # The foot defined in terms of inches
mile    5280 ft         # And the mile
```

A unit that ends with a '`-`' character is a prefix. If a prefix definition contains any '`/`' characters, be sure they are protected by parentheses. If you define '`half- 1/2`' then '`halfmeter`' would be equivalent to '`1 / (2 meter)`'.

9.3 Defining Nonlinear Units

Some unit conversions of interest are nonlinear; for example, temperature conversions between the Fahrenheit and Celsius scales cannot be done by simply multiplying by conversion factors.

When you give a linear unit definition such as '`inch 2.54 cm`' you are providing information that `units` uses to convert values in inches into primitive units of meters. For nonlinear units, you give a functional definition that provides the same information.

Nonlinear units are represented using a functional notation. It is best to regard this notation not as a function call but as a way of adding units to a number, much the same way that writing a linear unit name after a number adds units to that number. Internally, nonlinear units are defined by a pair of functions that convert to and from linear units in the data file, so that an eventual conversion to primitive units is possible.

Here is an example nonlinear unit definition:

```
tempF(x) units=[1;K] (x+(-32)) degF + stdtemp ; \
                     (tempF+(-stdtemp))/degF + 32
```

A nonlinear unit definition comprises a unit name, a dummy parameter name, two functions, and two corresponding units. The functions tell `units` how to convert to and from the new unit. In order to produce valid results, the arguments of these functions need to have the correct dimensions. To facilitate error checking, you may optionally indicate units for these arguments.

The definition begins with the unit name followed immediately (with no spaces) by a '`(`' character. In parentheses is the name of the parameter. Next is an optional specification of the units required by the functions in this definition. In the example above, the '`tempF`' function requires an input argument conformable with '`1`'. For normal nonlinear units definitions the forward function will always take a dimensionless argument. The inverse function requires an input argument conformable with '`K`'. In general the inverse function will need units that match the quantity measured by your nonlinear unit. The purpose of the expression in brackets to enable `units` to perform error checking on function arguments, and also to assign units to range and domain specifications, which are described later.

Next the function definitions appear. In the example above, the '`tempF`' function is defined by

```
tempF(x) = (x+(-32)) degF + stdtemp
```

This gives a rule for converting '`x`' in the units '`tempF`' to linear units of absolute temperature, which makes it possible to convert from tempF to other units.

In order to make conversions to Fahrenheit possible, you must give a rule for the inverse conversions. The inverse will be '`x(tempF)`' and its definition appears after a '`;`' character. In our example, the inverse is

```
x(tempF) = (tempF+(-stdtemp))/degF + 32
```

This inverse definition takes an absolute temperature as its argument and converts it to the Fahrenheit temperature. The inverse can be omitted by leaving out the '`;`' character, but then conversions to the unit will be impossible. If the inverse is omitted then the '`--check`' option will display a warning. It is up to you to calculate and enter the correct inverse function to obtain proper conversions. The '`--check`' option tests the inverse at one point and prints an error if it is not valid there, but this is not a guarantee that your inverse is correct.

If you wish to make synonyms for nonlinear units, you still need to define both the forward and inverse functions. Inverse functions can be obtained using the '`~`' operator. So to create a synonym for '`tempF`' you could write

```
fahrenheit(x) units=[1;K] tempF(x); ~tempF(fahrenheit)
```

You may define a function whose range and domain do not cover all of the real numbers. In this case `units` can handle errors better if you specify an appropriate range and domain. You specify the range and domain as shown below.

```
baume(d) units=[1;g/cm^3] domain=[0,130.5] range=[1,10] \
         (145/(145-d)) g/cm^3 ; (baume+-g/cm^3) 145 / baume
```

In this example the domain is specified after the '`domain=`' with the endpoints given in brackets. One of the end points can be omitted to get an interval that goes to infinity. So the range could be specified as nonnegative by writing '`range=[0,]`'. Both the range and domain are optional and can appear independently and in any order along with the '`units`' specification. The values in the range and domain are attached to the units given in the '`units`' specification. If you don't specify the units then the parameter inputs are reduced to primitive units for the numeric comparison to the values you give in the range or domain. In this case you should only use '`range`' or '`domain`' if the endpoints are zero and infinity.

Specifying the range and domain allows `units` to perform better error checking and give more helpful error messages when you invoke nonlinear units conversions outside of their bounds. It also enables the '`-c`' option to find a point in the domain to use for its point check of your inverse definition.

You may occasionally wish to define a function that operates on units. This can be done using a nonlinear unit definition. For example, the definition below provides conversion between radius and the area of a circle. This definition requires a length as input and produces an area as output, as indicated by the '`units=`' specification. Specifying the range as the nonnegative numbers can prevent cryptic error messages.

```
circlearea(r) units=[m;m^2] range=[0,]   pi r^2 ; sqrt(circlearea/pi)
```

Sometimes you may be interested in a piecewise linear unit such as many wire gauges. Piecewise linear units can be defined by specifying conversions to linear units on a list of points. Conversion at other points will be done by linear interpolation. A partial definition of zinc gauge is

```
zincgauge[in] 1 0.002, 10 0.02, 15 0.04, 19 0.06, 23 0.1
```

In this example, '`zincgauge`' is the name of the piecewise linear unit. The definition of such a unit is indicated by the embedded '`[`' character. After the bracket, you should indicate the units to be attached to the numbers in the table. No spaces can appear before the '`]`' character, so a definition like '`foo[kg meters]`' is illegal; instead write '`foo[kg*meters]`'. The definition of the unit consists of a list of pairs optionally separated by commas. This list defines a function for converting from the piecewise linear unit to linear units. The first item in each pair is the function argument; the second item is the value of the function at that argument (in the units specified in brackets). In this example, we define '`zincgauge`' at five points. For example, we set '`zincgauge(1)`' equal to '`0.002 in`'. Definitions like this may be more readable if written using continuation characters as

```
zincgauge[in] \
     1 0.002  \
    10 0.02   \
    15 0.04   \
    19 0.06   \
    23 0.1
```

With the preceding definition, the following conversion can be performed:

```
You have: zincgauge(10)
You want: in
    * 0.02
    / 50
You have: .01 inch
You want: zincgauge
    5
```

If you define a piecewise linear unit that is not strictly monotonic, then the inverse will not be well defined. If the inverse is requested for such a unit, `units` will return the smallest inverse. The '`--check`' option will print a warning if a non-monotonic piecewise linear unit is encountered.

9.4 Defining Unit List Aliases

Unit list aliases are treated differently from unit definitions, because they are a data entry shorthand rather than a true definition for a new unit. A unit list alias definition begins with '`!unitlist`' and includes the alias and the definition; for example, the aliases included in the standard units data file are

```
!unitlist   hms     hr;min;sec
!unitlist   time    year;day;hr;min;sec
!unitlist   dms     deg;arcmin;arcsec
!unitlist   ftin    ft;in;1|8 in
!unitlist   usvol   cup;3|4 cup;2|3 cup;1|2 cup;1|3 cup;1|4 cup;\
                    tbsp;tsp;1|2 tsp;1|4 tsp;1|8 tsp
```

Unit list aliases are only for unit lists, so the definition must include a '`;`'. Unit list aliases can never be combined with units or other unit list aliases, so the definition of '`time`' shown above could *not* have been shortened to '`year;day;hms`'. As usual, be sure to run `units --check` to ensure that the units listed in unit list aliases are conformable.

10 Numeric Output Format

By default, results of conversions are shown to eight significant figures; this can be changed with the '`--exponential`' and '`--output-format`' options. The former sets an exponential format (i.e., scientific notation) like that used in the original Unix `units` program; the latter allows the format to be given as that of the printf function in the ANSI C standard.

The format recognized with the '`--output-format`' option is a subset of that for printf(). Only a floating-point format of the form `%`[*flag*][*width*][`.`*precision*]*type* is allowed: it must begin with '`%`', and must end with a floating-point type specifier ('`E`' or '`e`' for scientific notation, '`f`' for fixed-point decimal, or '`G`' or '`g`' to specify the number of significant figures). The format specification may include one optional flag ('`+`', '`-`', '`#`', or a space), followed by an optional value for the minimum field width, and an optional precision specification that begins with a period (e.g., '`.6`'). In addition to the digits, the field width includes the decimal point, the exponent, and the sign if any of these are shown. A width specification is typically used with fixed-point decimal to have columns of numbers align at the decimal point; it normally is not useful with `units`. Non-floating-point type specifiers make no sense for `units`, and are forbidden.

The default format is '`%.8g`'; for greater precision, you could specify '`-o %.15g`'. The '`G`' and '`g`' formats use exponential format whenever the exponent would be less than -5, so the value 0.000013 displays as '`1.3e-005`'. If you prefer fixed-point display, you might specify '`-o %.8f`'; however, very small numbers may display very few significant figures, and for very small numbers, may show nothing but zeros.

See the documentation for printf() for more detailed descriptions of the format specification.

11 Localization

Some units have different values in different locations. The localization feature accommodates this by allowing a units data file to specify definitions that depend on the user's locale.

11.1 Locale

A locale is a subset of a user's environment that indicates the user's language and country, and some attendant preferences, such as the formatting of dates. The `units` program attempts to determine the locale from the POSIX setlocale function; if this cannot be done, `units` examines the environment variables `LC_CTYPE` and `LANG`. On POSIX systems, a locale is of the form *language_country*, where *language* is the two-character code from ISO 639-1 and *country* is the two-character code from ISO 3166-1; *language* is lower case and *country* is upper case. For example, the POSIX locale for the United Kingdom is `en_GB`.

On systems running Microsoft Windows, the value returned by setlocale() is different from that on POSIX systems; `units` attempts to map the Windows value to a POSIX value by means of a table in the file '`locale.map`' in the same directory, typically '`/usr/local/share/units`', as the default units data files. The file includes entries for many combinations of language and country, and can be extended to include other combinations. The '`locale.map`' comprises two tab-separated columns; each entry is of the form

Windows-locale *POSIX-locale*

where *POSIX-locale* is as described above, and *Windows-locale* typically spells out both the language and country. For example, the entry for the United States is

```
English_United States   en_US
```

You can force `units` to run in a desired locale by using the '`-l`' option.

In order to create unit definitions for a particular locale you begin a block of definitions in a unit datafile with '`!locale`' followed by a locale name. The '`!`' must be the first character on the line. The `units` program reads the following definitions only if the current locale matches. You end the block of localized units with '`!endlocale`'. Here is an example, which defines the British gallon.

```
!locale en_GB
gallon       4.54609 liter
!endlocale
```

11.2 Additional Localization

Sometimes the locale isn't sufficient to determine unit preferences. There could be regional preferences, or a company could have specific preferences. Though probably uncommon, such differences could arise with the choice of English customary units outside of English-speaking countries. To address this, `units` allows specifying definitions that depend on environment variable settings. The environment variables can be controled based on the current locale, or the user can set them to force a particular group of definitions.

A conditional block of definitions in a units data file begins with either '`!var`' or '`!varnot`' following by an environment variable name and then a space separated list of values. The leading '`!`' must appear in the first column of a units data file, and the conditional block is terminated by '`!endvar`'. Definitions in blocks beginning with '`!var`' are executed only if the environment variable is exactly equal to one of the listed values. Definitions in blocks beginning with '`!varnot`' are executed only if the environment variable does *not* equal any of the list values.

The inch has long been a customary measure of length in many places. The word comes from the latin *uncia* meaning "one twelfth," referring to its relationship with the foot. By the 20th century, the inch was officially defined in English-speaking countries relative to the yard, but until 1959, the yard differed slightly among those countries. In France the customary inch, which was displaced in 1799 by the meter, had a different length based on a french foot. These customary definitions could be accomodated as follows:

```
!var INCH_UNIT usa
yard          3600|3937 m
!endvar
```

```
!var INCH_UNIT canada
yard          0.9144 meter
!endvar
!var INCH_UNIT uk
yard          0.91439841 meter
!endvar
!var INCH_UNIT canada uk usa
foot          1|3 yard
inch          1|12 foot
!endvar
!var INCH_UNIT france
foot          144|443.296 m
inch          1|12 foot
line          1|12 inch
!endvar
!varnot INCH_UNIT usa uk france canada
!message Unknown value for INCH_UNIT
!endvar
```

When `units` reads the above definitions it will check the environment variable `INCH_UNIT` and load only the definitions for the appropriate section. If `INCH_UNIT` is unset or is not set to one of the four values listed then `units` will run the last block. In this case that block uses the '`!message`' command to display a warning message. Alternatively that block could set default values.

In order to create default values that are overridden by user settings the data file can use the '`!set`' command, which sets an environment variable *only if it is not already set*; these settings are only for the current `units` invocation and do not persist. So if the example above were preceded by '`!set INCH_UNIT france`' then this would make '`france`' the default value for `INCH_UNIT`. If the user had set the variable in the environment before invoking `units`, then `units` would use the user's value.

To link these settings to the user's locale you combine the '`!set`' command with the '`!locale`' command. If you wanted to combine the above example with suitable locales you could do by *preceding* the above definition with the following:

```
!locale en_US
!set INCH_UNIT usa
!endlocale
!locale en_GB
!set INCH_UNIT uk
!endlocale
!locale en_CA
!set INCH_UNIT canada
!endlocale
!locale fr_FR
!set INCH_UNIT france
!endlocale
!set INCH_UNIT france
```

These definitions set the overall default for `INCH_UNIT` to '`france`' and set default values for four locales appropriately. The overall default setting comes last so that it only applies when `INCH_UNIT` was not set by one of the other commands or by the user.

If the variable given after '`!var`' or '`!varnot`' is undefined then `units` prints an error message and ignores the definitions that follow. Use '`!set`' to create defaults to prevent this situation from arising. The '`-c`' option only checks the definitions that are active for the current environment and locale, so when adding new definitions take care to check that all cases give rise to a well defined set of definitions.

12 Environment Variables

The `units` program uses the following environment variables:

`HOME` Specifies the location of your home directory; it is used by `units` to find a personal units data file '`.units`'. On systems running Microsoft Windows, `units` tries to determine your home directory from the `HOMEDRIVE` and `HOMEPATH` environment variables if `HOME` does not exist.

`LC_CTYPE, LANG`
Checked to determine the locale if `units` cannot obtain it from the operating system. Sections of the standard units data file are specific to certain locales.

`MYUNITSFILE`
Specifies your personal units data file. If this variable exists, `units` uses its value rather than searching your home directory for '`.units`'. The personal units file will not be loaded if any data files are given using the '`-f`' option.

`PAGER` Specifies the pager to use for help and for displaying the conformable units. The help function browses the units database and calls the pager using the '`+n`'*n* syntax for specifying a line number. The default pager is `more`; `PAGER` can be used to specify alternatives such as `less`, `pg`, `emacs`, or `vi`.

`UNITS_ENGLISH`
Set to either '`US`' or '`GB`' to choose United States or British volume definitions, overriding the default from your locale.

`UNITSFILE`
Specifies the units data file to use (instead of the default). You can only specify a single units data file using this environment variable. If units data files are given using the '`-f`' option, the file specified by `UNITSFILE` will be not be loaded unless the '`-f`' option is given with the empty string ('`units -f ""`').

13 Unicode Support

The standard units data file is written in Unicode using the UTF-8 encoding. Portions of the file that are not plain ASCII begin with '`!utf8`' and end with '`!endutf8`'. As usual, the '`!`' must appear as the first character on the line. If a line of a data file contains byte sequences that are invalid UTF-8 or non-printing UTF-8 then `units` ignores the entire line.

When `units` runs it checks the locale to determine the character set. If UTF-8 is listed, then `units` reads the utf8 definitions. If any other character set is in use, then `units` works in plain ASCII without support for extended characters.

14 Readline Support

If the `readline` package has been compiled in, then when `units` is used interactively, numerous command line editing features are available. To check if your version of `units` includes `readline`, invoke the program with the '`--version`' option.

For complete information about `readline`, consult the documentation for the `readline` package. Without any configuration, `units` will allow editing in the style of emacs. Of particular use with `units` are the completion commands.

If you type a few characters and then hit ESC followed by `?` then `units` will display a list of all the units that start with the characters typed. For example, if you type *`metr`* and then request completion, you will see something like this:

```
You have: metr
metre             metriccup         metrichorsepower  metrictenth
metretes          metricfifth       metricounce       metricton
metriccarat       metricgrain       metricquart       metricyarncount
You have: metr
```

If there is a unique way to complete a unitname, you can hit the TAB key and `units` will provide the rest of the unit name. If `units` beeps, it means that there is no unique completion. Pressing the TAB key a second time will print the list of all completions.

15 Updating Currency Exchange Rates

The units program includes currency exchange rates and prices for some precious metals in the database. Of course, these values change over time, sometimes very rapidly, and `units` cannot provide real time values. To update the exchange rates run the `units_cur`, which rewrites the files containing the currency rates, typically '`/usr/local/share/units/currency.units`'. This program must be run with suitable permissions to write the file. To keep the rates updated automatically, it could be run by a cron job on a Unix-like system, or a similar scheduling program on a different system. Currency exchange rates are taken from Time Genie (`http://www.timegenie.com`) and precious metals pricing from Packetizer (`www.packetizer.com`). These sites update once per day, so there is no benefit in running the update script more often than daily. You can run `units_cur` with a filename specified on the command line and it will write the data to that file. If you give '`-`' for the file it will write to standard output.

16 Database Command Syntax

unit definition
: Define a regular unit.

prefix- definition
: Define a prefix.

funcname(*var*) units=[*in-units*,*out-units*] domain=[*x1*,*x2*] range=[*y1*,*y2*]
definition(var) ; *inverse(funcname)*
: Define a nonlinear unit or unit function. The three optional keywords `units=`, `range=` and `domain=` can appear in any order. The definition of the inverse is optional.

tabname[*out-units*] *pair-list*
: Define a piecewise linear unit. The pair list gives the points on the table listed in ascending order.

`!endlocale`
: End a block of definitions beginning with '`!locale`'

`!endutf8` End a block of definitions begun with '`!utf8`'

`!endvar` End a block of definitions begun with '`!var`' or '`!varnot`'

`!include` *file*
: Include the specified file.

`!locale` *value*
: Load the following definitions only of the locale is set to *value*.

`!message` *text*
: Display *text* when the database is read in, unless the quiet option ('`-q`') is enabled.

`!set` *variable value*
: Sets the environment variable, *variable*, to the specified value *only if* it is not already set.

`!unitlist` *alias definition*
: Define a unit list alias.

`!utf8` Load the following definitions only if `units` is running with UTF-8 enabled.

`!var` *variable value-list*
: Load the following definitions only if the environment variable, *variable* is set to one of the values listed on the space separated value list. If *variable* is not set then `units` prints an error message and ignores the following definitions.

`!varnot` *variable value-list*
: Load the following definitions only if the environment variable, *variable* is *not* set to one of the values listed on the space separated value list. If *variable* is not set then `units` prints an error message and ignores the following definitions.

Index

E

F

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Table of Contents