Engineering Calculator Documentation

Release 1.10

Ken Kundert

Mar 29, 2024

CONTENTS

Author: Ken Kundert <ec@nurdletech.com>
Date: 2022-12-27
Version: 1.10

This calculator is noteworthy in that it employs a stack model of computation (Reverse Polish Notation), it supports numbers with SI scale factors and units, and uses a text-only user interface.

CHAPTER

ONE

INSTALLING

Requires Python version 3.6 or later.

Install with:

```
pip install --user engineering-calculator
```

This installs *ec* into ~/.local/bin, which should be added to your path.

Unusually, there is also a man page. The Python install process no longer supports man pages, however you can download it from its Github repository. Place it in `~/.local/man/man1`.

CHAPTER

TWO

FEATURES

- Text-based interactive interface.
- Stack-based calculation model
- Full scientific/engineering calculator.
- Support SI scale factors and units on inputs and outputs.
- Supports hexadecimal, octal, and binary formats in both programmers and Verilog notation.
- Provides special functionality for electrical engineers.

CHAPTER

THREE

EXAMPLES

Invoke engineering calculator and compute the value of a resistor by dividing the voltage difference across the resistor by the current through it:

```
> ec
0: 2.5V 250mV - 1mA /
2.25k:
```

Here a hexadecimal number is converted to and from decimal:

```
> ec
0: 'hFF
255: vhex
'h00ff:
```

In this example, a frequency is converted to radians and saved into the variable *omega*, which is then used to compute the impedance of an inductor and capacitor:

```
> ec
0: 1MHz 2pi * =omega
6.2832M: 100nH *
628.32m: 1 10nF omega * /
15.915:
```

CHAPTER

FOUR

ISSUES

Please ask questions or report problems on Github.

CHAPTER

FIVE

DOCUMENTATION

5.1 Usage

ec [*options*] [*scripts* ...]

-i, --interactive Open an interactive session.

-s <file>, --startup <file> Run commands from file to initialize calculator before any script or interactive session is run, stack is cleared after it is run.

-c, --nocolor Do not use colors in the output.

-v, --verbose Narrate the execution of any scripts.

-V, --version Print the ec version information.

-h, --help Print the usage and exit.

5.2 Description

ec is a stack-based (RPN) engineering calculator with a text-based user interface that is intended to be used interactively.

If run with no arguments, an interactive session is started. If arguments are present, they are tested to see if they are filenames, and if so, the files are opened and the contents are executed as a script. If they are not file names, then the arguments themselves are treated as scripts and executed directly. The scripts are run in the order they are specified. In this case an interactive session would not normally be started, but if the interactive option is specified, it would be started after all scripts have been run.

The contents of *~/.ecrc*, *./.ecrc*, and the startup file will be run upon startup if they exist, and then the stack is cleared.

5.3 Stack

As you enter numbers they are pushed onto a stack. The most recent member of the stack is referred to as the *x* register and the second most recent is the *y* register. All other members of the stack are unnamed. Operators consume numbers off the stack to use as operands and then they push the results back on the stack. The operations are performed immediately and there is no use of parentheses to group calculations. Any intermediate results are stored on the stack until needed. For example,

```
4
6
+
```

In this case 4 gets pushed on to the stack first to become *x*. Then 6 gets pushed on to the stack to become *x*, which makes 4 *y*. Finally, + pulls both off the stack, sums them, and then pushes the result of 10 back onto the stack. The stack is left with only one number on it, 10.

After each line **ec** responds by printing the value of the *x* register. Thus the above example would actually look like this:

0: 4
4: 6
6: +
10:

The benefit of the stack is that it allows you to easily store temporary results while you perform your calculation. For example, to evaluate (34 - 61)*(23 - 56) you would use:

0: 34
34: 61
61: -
-27: 23
23: 56
56: -
-33: *
891:

Notice that you entered the numbers as you saw them in the formula you were evaluating, and there was no need to enter parentheses, however the operators were rearranged in order to express the precedence of the operations.

It is not necessary to type enter after each number or operator. You can combine them onto one line and just type enter when you would like to see the result:

0: 34 61 - 23 56 - *
891:

Furthermore, it is not necessary to type a space between a number and most operators. For example, the above could be entered as:

0: 34 61- 23 56- *
891:

You can print the entire stack using *stack*, and clear it using *clstack*. For example,

0: 1 2 3 stack
3
y: 2
x: 1
3: clstack
0: stack
0:

5.4 Real Numbers

Numbers can be entered using normal integer, floating point, and scientific notations. For example,

42
3.141592
5,439,749.97
2.998e8
13.80651e-24

In addition, you can also use the normal SI scale factors to represent either large or small numbers without using scientific notation.

Y	1e24 (yotta)
Z	1e21 (zetta)
E	1e18 (exa)
P	1e15 (peta)
T	1e12 (terra)
G	1e9 (giga)
M	1e6 (mega)
k, K	1e3 (kilo)
_	unity (1)
m	1e-3 (milli)
u	1e-6 (micro)
n	1e-9 (nano)
p	1e-12 (pico)
f	1e-15 (fempto)
a	1e-18 (atto)
z	1e-21 (zepto)
y	1e-24 (yocto)

For example, 10M represents 1e7 and 8.8p represents 8.8e-12.

Optionally, numbers can be combined with simple units. For example,

10KHz
3.16pF
2.5_V
4.7e-10F

Both units and scale factors are optional, which causes a natural ambiguity as to whether the first letter of a suffix is a scale factor or not. If the first letter is a valid scale factor, then it is assume to be a scale factor. In this way, '300K is treated as 300e3 rather than 300 Kelvin. If you intend the units without a scale factor, add the unit scale factor: '_'. Thus, use 300_K to enter 300 Kelvin.

In this case the units must be simple identifiers (must not contain special characters). For complex units, such as "rads/s", or for numbers that do not have scale factors, it is possible to attach units to a number in the *x* register by entering a quoted string.

0: 6.626e-34
662.6e-36: "J-s"
662.6e-36 J-s: 50k "V/V"
50 KV/V:

The dollar sign ($) is a special unit that is given before the number.

$100K

Numbers my also contain commas as digit separators, which are ignored.

$200,000.00

The dollar sign ($) is a special unit that is given before the number.

ec takes a conservative approach to units. You can enter them and it remembers them, but they do not survive any operation where the resulting units would be in doubt. In this way it displays units when it can, but should never display incorrect or misleading units. For example:

0: 100MHz
100 MHz: 2pi*
628.32M:

You can display real numbers using one of four available formats, *fix*, *sci*, *eng*, or *si*. These display numbers using fixed point notation (a fixed number of digits to the right of the decimal point), scientific notation (a mantissa and an exponent), engineering notation (a mantissa and an exponent, but the exponent is constrained to be a multiple of 3), and SI notation (a mantissa and a SI scale factor). You can optionally give an integer immediately after the display mode to indicate the desired precision. For example,

0: 10,000
10K: fix2
10,000.00: sci3
1.000e+04: eng2
10.0e+03: si4
10K: 2pi*
62.832K:

Notice that scientific and engineering notations always displays the specified number of digits whereas SI notation suppresses zeros at the end of the number.

When displaying numbers using SI notation, **ec** does not use the full range of available scale factors under the assumption that the largest and smallest would be unfamiliar to most people. For this reason, **ec** only uses the most common scale factors when outputting numbers (T, G, M, K, m, u, n, p, f, a).

5.5 Integers

You can enter integers in either hexadecimal (base 16), decimal (base 10), octal (base 8), or binary (base 2). You can use either programmers notation (leading 0) or Verilog notation (leading ‘) as shown in the examples below:

0xFF	hexadecimal
99	decimal
0o77	octal
0b1101	binary
‘hFF	Verilog hexadecimal
‘d99	Verilog decimal
‘o77	Verilog octal
‘b1101	Verilog binary

Internally, **ec** represents all numbers as double-precision real numbers. To display them as decimal integers, use *fix0*. However, you can display the numbers in either base 16 (hexadecimal), base 10 (decimal), base 8 (octal) or base 2 (binary) by setting the display mode. Use either *hex*, *fix0*, *oct*, *bin*, *vhex*, *vdec*, *voct*, or *vbin*. In each of these cases

the number is rounded to the closest integer before it is displayed. Add an integer after the display mode to control the number of digits. For example:

0: 1000
1K: hex
0x3b8: hex8
0x000003b8: hex0
0x3b8: voct
'o1750:

5.6 Complex Numbers

ec provides limited support for complex numbers. Two imaginary constants are available that can be used to construct complex numbers, *j* and *j2pi*. In addition, two functions are available for converting complex numbers to real, *mag* returns the magnitude and *ph* returns the phase. They are unusual in that they do not replace the value in the *x* register with the result, instead they simply push either the magnitude of phase into the *x* register, which pushes the original complex number into the *y* register. For example,

0: 1 j +
1 + j: mag
1.4142: pop
1 + j: ph
45 degs: stack
    y: 1 + j
    x: 45 degs
45 degs:

You can also add the imaginary unit to real number constants. For example,

0: j10M
j10M: -j1u *
10:

Only a small number of functions actually support complex numbers; currently only *exp* and *sqrt*. However, most of the basic arithmetic operators support complex numbers.

5.7 Constants

ec provides several useful mathematical and physical constants that are accessed by specifying them by name. Several of the constants have both MKS and CGS forms (ec uses ESU-CGS). You can set which version you want by setting the desired unit system as follows:

0: mks
0: h
662.61e-36 J-s: k
13.806e-24 J/K: cgs
13.806e-24 J/K: h
6.6261e-27 erg-s: k
138.06 aerg/K:

Notice that the unit-system is sticky, meaning that it remains in force until explicitly changed. 'mks' is the default unit system.

The physical constants are given in base units (meters, grams, seconds). For example, the mass of an electron is given in grams rather than kilograms as would be expected for MKS units. Similarly, the speed of light is given in meters per second rather than centimeters per second as would be expected of CGS units. This is necessary so that numbers are not displayed with two scale factors (ex. 1 mkg). Thus, it may be necessary for you to explicitly convert to kg (MKS) or cm (CGS) before using values in formulas that are tailored for one specific unit system.

The 2014 NIST values are used. The available constants include:

pi	the ratio of a circle's circumference to its diameter
2pi	the ratio of a circle's circumference to its radius
rt2	square root of two
0C	0 Celsius in Kelvin
j	imaginary unit (square root of 1)
j2pi	j2
k	Boltzmann constant
h	Planck constant
q	elementary charge (the charge of an electron)
c	speed of light in a vacuum
eps0	permittivity of free space
mu0	permeability of free space
Z0	Characteristic impedance of free space
hbar	Reduced Planck constant
me	rest mass of an electron
mp	mass of a proton
mn	mass of a neutron
mh	mass of a hydrogen atom
amu	unified atomic mass unit
G	universal gravitational constant
g	earth gravity
Rinf	Rydberg constant
sigma	Stefan-Boltzmann constant
alpha	Fine structure constant
R	molar gas constant
NA	Avogadro Number
rand	random number between 0 and 1

As an example of using the predefined constants, consider computing the thermal voltage, kT/q.

0: k 27 0C + * q/ "V"
25.865 mV:

5.8 Variables

You can store the contents of the *x* register to a variable by using an equal sign followed immediately by the name of the variable:

```
=«name»
```

where «name» represents the desired name of the variable. It must be a simple identifer and must be immediately adjacent to the =.

To recall it, simply use the name:

```
«name»
```

For example,

0: 100MHz =freq
100 MHz: 2pi* “rads/s” =omega
628.32 Mrads/s: 1pF =cin
1 pF: 1 omega cin* /
1.5915K:

You can display all known variables using *vars*. If you did so immediately after entering the lines above, you would see:

1.5915K: vars
Rref: 50 Ohms
cin: 1 pF
freq: 100 MHz
omega: 628.32 Mrads/s

Choosing a variable name that is the same as a one of a built-in command or constant causes the built-in name to be overridden. Be careful when doing this as once a built-in name is overridden it can no longer be accessed.

Notice that a variable *Rref* exists that you did not create. This is a predefined variable that is used in dBm calculations. You are free to change its value if you like.

5.9 User-Defined Functions

You can define functions in the following way:

```
( ... )«name»
```

Here ‘(’ starts the function definition and ‘)«name»’ ends it. «name» represents the desired name of the function. It must be a simple identifer and must be immediately adjacent to the). The ‘...’ represents a sequence of calculator actions.

For example:

0: (2pi * “rads/s”)to_omega
0: (2pi / “Hz”)to_freq
0: 100MHz
100 MHz: to_omega
628.32 Mrads/s: to_freq

100 MHz:

The actions entered while defining the function are not evaluated until the function itself is evaluated.

Once defined, you can review your function with the *vars* command. It shows both the variable and the function definitions:

Rref: 50 Ohms
to_freq: (2pi / "Hz")
to_omega: (2pi * "rads/s")

The value of the functions are delimited with parentheses.

5.10 Unit Conversions

You can perform unit conversions using:

>«units»

where «units» are the desired units. «units» must be a simple identifier though the first character is allowed to be one of a small number of special characters that typically begin units. Specifically, $, °, Å, , or ș``. «units» must be immediately adjacent to the >. If the value in the *x* register has units, the value will be converted to the new units. If the value does not have units, its units will be set to «units».

For example, to convert grams to pounds and back:

0: 100kg
100 kg: >lbs
220.46 lbs: >g
100 kg:

The following example is a little contrived to show two things. First, a value with no units gets assigned the specified units when subject to a conversion. Second, several names are known for the same units and the name assigned to the result is the name specified on the convert command.

0: 100
100: >kg
100 kg: >lb
220.46 lb:

Converters are provides for the following units:

Temperature:

K:	K, F °F, R °R
C °C:	K, C °C, F °F, R °R

Distance:

m:	km, m, cm, mm, um m micron, nm, Å angstrom, mi mile miles, in inch inches

Mass/Weight:

g:	oz, lb lbs

Time:

s:	sec second seconds, min minute minutes, hour hours hr, day days

Bits/Bytes:

b:	B

Bitcoin:

BTC btc :	USD usd $, sats sat ṣ
sats sat ṣ:	USD usd $, BTC btc

The conversions can occur between a pair of units, one from the first column and one from the second. They do not occur when both units are only in the second column. So for example, it is possible to convert between *g* and *lbs*, but not between *oz* and *lb*. However, if you notice, the units in the second column are grouped using commas. A set of units within commas are considered equivalent, meaning that there are multiple names for the same underlying unit. For example, *in*, *inch*, and *inches* are all considered equivalent. You can convert between equivalent units even though both are found in the second column.

Bitcoin conversions are performed by accessing quotes from coingecko.com. You must have an internet connection for this feature to work.

For example:

0: 1BTC
1 BTC: >$
$46,485.00:

You can use user-defined functions to create functions that create units directly. For example, here are function definitions for converting bitcoin and temperatures that you can put in your ~/.ecrc file:

```
# bitcoin
(>    )tb    # convert unitless number to bitcoin
(>  >$)btd   # convert bitcoin to dollars
(>  >ṣ)bts   # convert bitcoin to satoshis
(>ṣ    )ts    # convert unitless number to satoshis
(>ṣ  >)stb   # convert satoshis to bitcoin
(>ṣ  >)stb   # convert satoshis to bitcoin
(>ṣ  >$)std   # convert bitcoin to dollars
(>$  >)dtb   # convert dollars to bitcoin
(>$  >ṣ)dts   # convert dollars to satoshis

# temperature
(>°C   )tc    # convert unitless number to Celsius
(>°C >K)ctk   # convert Celsius to Kelvin
(>°C >°F)ctf  # convert Celsius to Fahrenheit
(>°F    )tf   # convert unitless number to Fahrenheit
(>°F >K)ftk   # convert Fahrenheit to Kelvin
(>°F >°C)ftc  # convert Fahrenheit to Celsius
(>K  >°C)ktc  # convert Kelvin to Celsius
(>K  >°F)ktf  # convert Kelvin to Fahrenheit
```

With these function, you can convert a simple number (without units) directly to the desired units:

0: 1 btd
$46,485.00:

5.11 Comments

Any text that follows a # is ignored. In this way you can add documentation to initialization files and scripts, as shown in the next few sections.

5.12 Help

You can use *help* to get a listing of the various features available in EC along with a short summary of each feature. For more detailed information, you can use '?'. If you use '?' alone you will get a list of all available help topics. If you use '?<*topic*>' where *topic* us either a symbol or a name, you will get a detailed description of that topic.

5.13 Initialization

At start up **ec** reads and executes commands from files. It first tries '~/.ecrc' and runs any commands it contains if it exists. It then tries './.ecrc' if it exists. Finally it runs the startup file specified on the command line (with the **-s** or **–startup** option). It is common to put your generic preferences in '~/.exrc'. For example, if your are an astronomer with a desire for high precision results, you might use:

```
# initialization file for ec (engineering calculator)
eng6
6.626070e-27 "erg-s" =h       # Planck's constant in CGS units
1.054571800e-27 "erg-s" =hbar # Reduced Planck's constant in CGS units
1.38064852e-16 "erg/K" =k     # Boltzmann's constant in CGS units
```

This tells **ec** to use 6 digits of resolution and redefines *h* and *hbar* so that they are given in CGS units. The redefining of the names *h*, *hbar*, and *k* would normally cause **ec** to print a warning, but such warnings are suppressed when reading initialization files and scripts.

After all of the startup files have been processed, the stack is cleared.

A typical initialization script (~/.ecrc) for a circuit designer might be:

```
# Initialize Engineering Calculator
27 "C" =T               # ambient temperature
(k T 0C + * q/ "V")vt   # thermal voltage
(2pi* "rads/s")tw       # to omega - converts Hertz to rads/s
(2pi/ "Hz")tf           # to freq - converts rads/s to Hertz
```

5.14 Scripting

Command line arguments are evaluated as if they were typed into an interactive session with the exception of filename arguments. If an argument corresponds to an existing file, the file treated as a script, meaning it is is opened its contents are evaluated. Otherwise, the argument itself is evaluate (often it needs to be quoted to protect its contents from being interpreted by the shell). When arguments are given the calculator by default does not start an interactive session. For example: to compute an RC time constant you could use:

$ ec 22k 1pF*
22n

Notice that the * in the above command is interpreted as glob character, which is generally not what you want, so it is often best to quote the script:

$ ec '22k 1pF*'
22n

Only the calculator commands would be quoted in this manner. If you included a file name on the command line to run a script, it would have to be given alone. For example, assume that the file 'bw' exists and contains '* 2pi* recip "Hz"'. This is a script that assumes that the value of R and C are present in the *x* and *y* resisters, and then computes the 3dB bandwith of the corresponding RC filter. You could run the script with:

$ ec '22k 1pF' bw
7.2343 MHz

Normally **ec** only prints the value of the *x* register and only as it exits. It is possible to get more control of the output using back-quoted strings. For example:

$ ec '`Hello world!`'
Hello world!
0

Whatever is found within back-quotes is printed to the output. Notice that the value of the *x* register is also output, which may not be desired when you are generating your own output. You can stop the value of the *x* register from being printed by finishing with the *quit* command, which tells **ec** to exit immediately:

$ ec '`Hello world!` quit'
Hello world!

You can add the values of registers and variables to your print statements. *$N* prints out the value of register *N*, where 0 is the *x* register, 1 is the *y* register, etc. *$name* will print the value of a variable with the given name. Alternatively, you can use *${N}* and *${name}* to disambiguate the name or number. To print a dollar sign, use *$$*. To print a newline or a tab, use *\n* and *\t*. For example,

0: 100MHz =freq
100 MHz: 2pi* "rads/s"
628.32 Mrads/s: `$freq corresponds to $0.`
100 MHz corresponds to 628.32 Mrads/s.
628.32 Mrads/s:

To illustrate its use in a script, assume that a file named *lg* exists and contains a calculation for the loop gain of a PLL:

```
# computes and displays loop gain of a frequency synthesizer
# x register is taken to be frequency
=freq
88.3u "V/per" =Kdet  # gain of phase detector
9.07G "Hz/V" =Kvco   # gain of voltage controlled oscillator
```

(continues on next page)

(continued from previous page)

```
2 =M                # divide ratio of divider at output of VCO
8 =N                # divide ratio of main divider
2 =F                # divide ratio of prescalar
freq 2pi* "rads/s" =omega
Kdet Kvco* omega/ M/ =a
N F* =f
a f* =T
\`Open loop gain = $a\\nFeedback factor = $f\\nLoop gain = $T\`
quit
```

When reading scripts from a file, the '#' character introduces a comment. It and anything that follows is ignored until the end of the line.

Notice that the script starts by saving the value in the *x* register to the variable *freq*. This script would be run as:

$ ec 1KHz lg
Open loop gain = 63.732
Feedback factor = 16
Loop gain = 1.0197K

The first argument does not correspond to a file, so it is executed as a script. It simply pushes 1KHz onto the stack. The second argument does correspond to a file, so its contents are executed. The script ends with a print command, so the results are printed to standard output as the script terminates.

One issue with command line scripting that you need to be careful of is that if an argument is a number with a leading minus sign it will be mistaken to be a command line option. To avoid this issue, specify the number without the minus sign and follow it with *chs*. Alternatively, you can embed the number in quotes but add a leading space. For example,

$ ec -30 dbmv
ec: -30 dbmv: unknown option.
$ ec 30 chs dbmv
10 mV
$ ec ' -30' dbmv
10 mV

5.15 Initialization Scripts

You can use scripts to preload in a set of useful constants and function that can then be used in interactive calculations. To do so, use the **-i** or *–interactive* command line option. For example, replace the earlier 'lg' script with the following:

88.3u "V/per" =Kdet
9.07G "Hz/V" =Kvco
2 =M
8 =N
2 =F
(N F* recip)f
(2pi * Kdet * Kvco* M*)a
(a f*)T
clstack

Now run:

$ ec -i lg

0: 1kHz T
629.01M:

Doing so runs lg, which loads values into the various variables, and then they can be accessed in further calculations.

Notice that the script ends with *clstack* so that you start fresh in your interactive session. It simply clears the stack so that the only effect of the script is to initialize the variables. Using **-s** or **–startup** does this for you automatically.

Alternatively, you can put the constants you wish to predeclare in *./.ecrc*, in which case they are automatically loaded whenever you invoke **ec** in the directory that contains the file. Similarly, placing constants in *~/.ecrc* causes them to be declared for every invocation of **ec**.

5.16 Errors

If an error occurs on a line, an error message is printed and the stack is restored to the values it had before the line was entered. So it is almost as if you never typed the line in at all. The exception being that any variables or modes that are set on the line before the error occurred are retained. For example,

0: 1KOhms =r
1 KOhms: 100MHz =freq 1pF = c
=: unrecognized
1 KOhms: stack
x: 1 KOhms
1 KOhms: vars
Rref: 50 Ohms
freq: 100MHz
r: 1 KOhms

The error occurred when trying to assign a value to *c* because a space was accidentally left between the equal sign and the variable name. Notice that 100MHz was saved to the variable *freq*, but the stack was restored to the state it had before the offending line was entered.

5.17 Operator, Function, Number and Command Reference

In the following descriptions, optional values are given in brackets ([]) and values given in angle brackets (<>) are not to be taken literally (you are expected to choose a suitable value). For example "fix[<*N*>]" can represent "fix" or "fix4", but not "fixN".

For each action that changes the stack a synopsis of those changes is given in the form of two lists separated by =>. The list on the left represents the stack before the action is applied, and the list on the right represents the stack after the action was applied. In both of these lists, the *x* register is given first (on the left). Those registers that are involved in the action are listed explicitly, and the rest are represented by `...`. In the before picture, the names of the registers involved in the action are simply named. In the after picture, the new values of the registers are described. Those values represented by `...` on the right side of => are the same as represented by `...` on the left, though they may have moved. For example:

```
x, y, ... => x+y, ...
```

This represents addition. In this case the values in the *x* and *y* registers are summed and placed into the *x* register. All other values move to the left one place.

5.17.1 Arithmetic Operators

+: addition

The values in the *x* and *y* registers are popped from the stack and the sum is placed back on the stack into the *x* register.

```
x, y, ... → x+y, ...
```

-: subtraction

The values in the *x* and *y* registers are popped from the stack and the difference is placed back on the stack into the *x* register.

```
x, y, ... → x-y, ...
```

*: multiplication

The values in the *x* and *y* registers are popped from the stack and the product is placed back on the stack into the *x* register.

```
x, y, ... → x*y, ...
```

/: true division

The values in the *x* and *y* registers are popped from the stack and the quotient is placed back on the stack into the *x* register. Both values are treated as real numbers and the result is a real number. So

0: 1 2/

500m:

```
x, y, ... → y/x, ...
```

//: floor division

The values in the *x* and *y* registers are popped from the stack, the quotient is computed and then converted to an integer using the floor operation (it is replaced by the largest integer that is smaller than the quotient), and that is placed back on the stack into the *x* register. So

0: 1 2//

0:

```
x, y, ... → y//x, ...
```

%: modulus

The values in the *x* and *y* registers are popped from the stack, the quotient is computed and the remainder is placed back on the stack into the *x* register. So

0: 14 3%

2:

In this case 2 is the remainder because 3 goes evenly into 14 three times, which leaves a remainder of 2.

```
x, y, ... → y%x, ...
```

chs: change sign

The value in the *x* register is replaced with its negative.

```
x, ... → x, ...
```

`recip`: reciprocal

The value in the *x* register is replaced with its reciprocal.

```
x, ... → 1/x, ...
```

`ceil`: round towards positive infinity

The value in the *x* register is replaced with its value rounded towards infinity (replaced with the smallest integer greater than its value).

```
x, ... → ceil(x), ...
```

`floor`: round towards negative infinity

The value in the *x* register is replaced with its value rounded towards negative infinity (replaced with the largest integer smaller than its value).

```
x, ... → floor(x), ...
```

`!`: factorial

The value in the *x* register is replaced with the factorial of its value rounded to the nearest integer.

```
x, ... → x!, ...
```

`%chg`: percent change

The values in the *x* and *y* registers are popped from the stack and the percent difference between *x* and *y* relative to *y* is pushed back into the *x* register.

```
x, y, ... → 100*(x-y)/y, ...
```

`||`: parallel combination

The values in the *x* and *y* registers are popped from the stack and replaced with the reciprocal of the sum of their reciprocals. If the values in the *x* and *y* registers are both resistances, both elastances, or both inductances, then the result is the resistance, elastance or inductance of the two in parallel. If the values are conductances, capacitances or susceptances, then the result is the conductance, capacitance or susceptance of the two in series.

```
x, y, ... → 1/(1/x+1/y), ...
```

5.17.2 Powers, Roots, Exponentials and Logarithms

`**`: raise y to the power of x

The values in the *x* and *y* registers are popped from the stack and replaced with the value of *y* raised to the power of *x*.

```
x, y, ... → y**x, ...
```

aliases: pow,ytox

`exp`: natural exponential

The value in the *x* register is replaced with its exponential. Supports a complex argument.

```
x, ... → exp(x), ...
```

alias: powe

`ln`: natural logarithm

The value in the *x* register is replaced with its natural logarithm. Supports a complex argument.

```
x, ... → ln(x), ...
```

alias: loge

`pow10`: raise 10 to the power of x

The value in the *x* register is replaced with 10 raised to *x*.

```
x, ... → 10**x, ...
```

alias: 10tox

`log`: base 10 logarithm

The value in the *x* register is replaced with its common logarithm.

```
x, ... → log(x), ...
```

aliases: log10,lg

`pow2`: raise 2 to the power of x

The value in the *x* register is replaced with 2 raised to *x*.

```
x, ... → 2**x, ...
```

alias: 2tox

`log2`: base 2 logarithm

The value in the *x* register is replaced with its base 2 logarithm.

```
x, ... → log2(x), ...
```

alias: lb

`sqr`: square

The value in the *x* register is replaced with its square.

```
x, ... → x**2, ...
```

`sqrt`: square root

The value in the *x* register is replaced with its square root.

```
x, ... → sqrt(x), ...
```

alias: rt

`cbrt`: cube root

The value in the *x* register is replaced with its cube root.

```
x, ... → cbrt(x), ...
```

5.17.3 Trigonometric Functions

`sin`: trigonometric sine

The value in the *x* register is replaced with its sine.

```
x, ... → sin(x), ...
```

`cos`: trigonometric cosine

The value in the *x* register is replaced with its cosine.

```
x, ... → cos(x), ...
```

`tan`: trigonometric tangent

The value in the *x* register is replaced with its tangent.

```
x, ... → tan(x), ...
```

`asin`: trigonometric arc sine

The value in the *x* register is replaced with its arc sine.

```
x, ... → asin(x), ...
```

`acos`: trigonometric arc cosine

The value in the *x* register is replaced with its arc cosine.

```
x, ... → acos(x), ...
```

`atan`: trigonometric arc tangent

The value in the *x* register is replaced with its arc tangent.

```
x, ... → atan(x), ...
```

`rads`: use radians

Switch the trigonometric mode to radians (functions such as *sin*, *cos*, *tan*, and *ptor* expect angles to be given in radians; functions such as *arg*, *asin*, *acos*, *atan*, *atan2*, and *rtop* should produce angles in radians).

`degs`: use degrees

Switch the trigonometric mode to degrees (functions such as *sin*, *cos*, *tan*, and *ptor* expect angles to be given in degrees; functions such as *arg*, *asin*, *acos*, *atan*, *atan2*, and *rtop* should produce angles in degrees).

5.17.4 Complex and Vector Functions

`abs`: magnitude of complex number

The absolute value of the number in the *x* register is pushed onto the stack if it is real. If the value is complex, the magnitude is pushed onto the stack.

```
x, ... → abs(x), x, ...
```

alias: mag

`arg`: phase of complex number

The argument of the number in the *x* register is pushed onto the stack if it is complex. If the value is real, zero is pushed onto the stack.

```
x, ... → arg(x), x, ...
```

alias: ph

`hypot`: hypotenuse

The values in the *x* and *y* registers are popped from the stack and replaced with the length of the vector from the origin to the point (*x*, *y*).

```
x, y, ... → sqrt(x**2+y**2), ...
```

alias: len

`atan2`: two-argument arc tangent

The values in the *x* and *y* registers are popped from the stack and replaced with the angle of the vector from the origin to the point.

```
x, y, ... → atan2(y,x), ...
```

alias: angle

`rtop`: convert rectangular to polar coordinates

The values in the *x* and *y* registers are popped from the stack and replaced with the length of the vector from the origin to the point (*x*, *y*) and with the angle of the vector from the origin to the point (*x*, *y*).

```
x, y, ... → sqrt(x**2+y**2), atan2(y,x), ...
```

`ptor`: convert polar to rectangular coordinates

The values in the *x* and *y* registers are popped from the stack and interpreted as the length and angle of a vector and are replaced with the coordinates of the end-point of that vector.

```
x, y, ... → x*cos(y), x*sin(y), ...
```

5.17.5 Hyperbolic Functions

`sinh`: hyperbolic sine

The value in the *x* register is replaced with its hyperbolic sine.

```
x, ... → sinh(x), ...
```

`cosh`: hyperbolic cosine

The value in the *x* register is replaced with its hyperbolic cosine.

```
x, ... → cosh(x), ...
```

`tanh`: hyperbolic tangent

The value in the *x* register is replaced with its hyperbolic tangent.

```
x, ... → tanh(x), ...
```

`asinh`: hyperbolic arc sine

The value in the *x* register is replaced with its hyperbolic arc sine.

```
x, ... → asinh(x), ...
```

`acosh`: hyperbolic arc cosine

The value in the *x* register is replaced with its hyperbolic arc cosine.

```
x, ... → acosh(x), ...
```

`atanh`: hyperbolic arc tangent

The value in the *x* register is replaced with its hyperbolic arc tangent.

```
x, ... → atanh(x), ...
```

5.17.6 Decibel Functions

`db`: convert voltage or current to dB

The value in the *x* register is replaced with its value in decibels. It is appropriate to apply this form when converting voltage or current to decibels.

```
x, ... → 20*log(x), ...
```

aliases: db20,v2db,i2db

`adb`: convert dB to voltage or current

The value in the *x* register is converted from decibels and that value is placed back into the *x* register. It is appropriate to apply this form when converting decibels to voltage or current.

```
x, ... → 10**(x/20), ...
```

aliases: db2v,db2i

`db10`: convert power to dB

The value in the *x* register is converted from decibels and that value is placed back into the *x* register. It is appropriate to apply this form when converting power to decibels.

```
x, ... → 10*log(x), ...
```

alias: p2db

`adb10`: convert dB to power

The value in the *x* register is converted from decibels and that value is placed back into the *x* register. It is appropriate to apply this form when converting decibels to voltage or current.

```
x, ... → 10**(x/10), ...
```

alias: db2p

`vdbm`: convert peak voltage to dBm

The value in the *x* register is expected to be the peak voltage of a sinusoid that is driving a load resistor equal to *Rref* (a predefined variable). It is replaced with the power delivered to the resistor in decibels relative to 1 milliwatt.

```
x, ... → 30+10*log10((x**2)/(2*Rref)), ...
```

alias: v2dbm

`dbmv`: dBm to peak voltage

The value in the *x* register is expected to be a power in decibels relative to one milliwatt. It is replaced with the peak voltage of a sinusoid that would be needed to deliver the same power to a load resistor equal to *Rref* (a predefined variable).

```
x, ... → sqrt(2*10**(x - 30)/10)*Rref), ...
```

alias: dbm2v

`idbm`: peak current to dBm

The value in the *x* register is expected to be the peak current of a sinusoid that is driving a load resistor equal to *Rref* (a predefined variable). It is replaced with the power delivered to the resistor in decibels relative to 1 milliwatt.

```
x, ... → 30+10*log10(((x**2)*Rref/2), ...
```

alias: i2dbm

`dbmi`: dBm to peak current

The value in the *x* register is expected to be a power in decibels relative to one milliwatt. It is replaced with the peak current of a sinusoid that would be needed to deliver the same power to a load resistor equal to *Rref* (a predefined variable).

```
x, ... → sqrt(2*10**(x - 30)/10)/Rref), ...
```

alias: dbm2i

5.17.7 Constants

`pi`: the ratio of a circle's circumference to its diameter

The value of (3.141592...) is pushed on the stack into the *x* register.

```
... → , ...
```

alias:

`2pi`: the ratio of a circle's circumference to its radius

2 (6.283185...) is pushed on the stack into the *x* register.

```
... → 2, ...
```

aliases: tau,,2

`rt2`: square root of two

2 (1.4142...) is pushed on the stack into the *x* register.

```
... → 2, ...
```

`0C`: 0 Celsius in Kelvin

Zero celsius in kelvin (273.15 K) is pushed on the stack into the *x* register.

```
... → 0C, ...
```

`j`: imaginary unit (square root of 1)

The imaginary unit (square root of -1) is pushed on the stack into the *x* register.

```
... → j, ...
```

`j2pi`: j2

2 times the imaginary unit (j6.283185...) is pushed on the stack into the *x* register.

```
... → j*2*pi, ...
```

aliases: jtau,j,j2

`k`: Boltzmann constant

The Boltzmann constant (R/NA or $1.38064852 \times 10^{23}$ J/K [mks] or $1.38064852 \times 10^{16}$ erg/K [cgs]) is pushed on the stack into the *x* register.

```
... → k, ...
```

`h`: Planck constant

The Planck constant *h* (6.626070×10^{34} J-s [mks] or 6.626070×10^{27} erg-s [cgs]) is pushed on the stack into the *x* register.

```
... → h, ...
```

`q`: elementary charge (the charge of an electron)

The elementary charge (the charge of an electron or $1.6021766208 \times 10^{19}$ C [mks] or $4.80320425 \times 10^{10}$ statC [cgs]) is pushed on the stack into the *x* register.

```
... → q, ...
```

`c`: speed of light in a vacuum

The speed of light in a vacuum (2.99792458×10^8 m/s) is pushed on the stack into the *x* register.

```
... → c, ...
```

`eps0`: permittivity of free space

The permittivity of free space ($8.854187817 \times 10^{12}$ F/m [mks] or 1/4 [cgs]) is pushed on the stack into the *x* register.

```
... → eps0, ...
```

`mu0`: permeability of free space

The permeability of free space (4×10^7 H/m [mks] or $4/c^2$ s^2/m^2 [cgs]) is pushed on the stack into the *x* register.

```
... → mu0, ...
```

`Z0`: Characteristic impedance of free space

The characteristic impedance of free space (376.730313461) is pushed on the stack into the *x* register.

```
... → Z0, ...
```

`hbar`: Reduced Planck constant

The reduced Planck constant ($1.054571800 \times 10^{34}$ J-s [mks] or $1.054571800 \times 10^{27}$ erg-s [cgs]) is pushed on the stack into the *x* register.

```
... → , ...
```

alias:

`me`: rest mass of an electron

The rest mass of an electron ($9.10938356 \times 10^{28}$ g) is pushed on the stack into the *x* register.

```
... → me, ...
```

`mp`: mass of a proton

The mass of a proton ($1.672621898 \times 10^{24}$ g) is pushed on the stack into the *x* register.

```
... → mp, ...
```

`mn`: mass of a neutron

The mass of a neutron ($1.674927471 \times 10^{24}$ g) is pushed on the stack into the *x* register.

```
... → mn, ...
```

`mh`: mass of a hydrogen atom

The mass of a hydrogen atom ($1.6735328115 \times 10^{24}$ g) is pushed on the stack into the *x* register.

```
... → mh, ...
```

`amu`: unified atomic mass unit

The unified atomic mass unit ($1.660539040 \times 10^{24}$ g) is pushed on the stack into the *x* register.

```
... → amu, ...
```

`G`: universal gravitational constant

The universal gravitational constant (6.6746×10^{14} $m^3/g\text{-}s^2$) is pushed on the stack into the *x* register.

```
... → G, ...
```

`g`: earth gravity

The standard acceleration at sea level due to gravity on earth (9.80665 m/s^2)) is pushed on the stack into the *x* register.

```
... → g, ...
```

`Rinf`: Rydberg constant

The Rydberg constant (10973731 m^1) is pushed on the stack into the *x* register.

```
... → Ry, ...
```

`sigma`: Stefan-Boltzmann constant

The Stefan-Boltzmann constant (5.670367×10^8 W/m^2K^4) is pushed on the stack into the *x* register.

```
... → sigma, ...
```

`alpha`: Fine structure constant

The fine structure constant (7.2973525664e-3) is pushed on the stack into the *x* register.

```
... → alpha, ...
```

`R`: molar gas constant

The molar gas constant (8.3144598 J/mol-K [mks] or 83.145 Merg/deg-mol [cgs]) is pushed on the stack into the *x* register.

```
... → R, ...
```

`NA`: Avogadro Number

Avogadro constant ($6.022140857 \times 10^{23}$ mol^1) is pushed on the stack into the *x* register.

```
... → NA, ...
```

`mks`: use MKS units for constants

Switch the unit system for constants to MKS or SI.

`cgs`: use ESU CGS units for constants

Switch the unit system for constants to ESU CGS.

5.17.8 Numbers

`«N[.M][S][U]»`: a real number

The number is pushed on the stack into the *x* register. *N* is the integer portion of the mantissa and *M* is an optional fractional part. *S* is a letter that represents an SI scale factor. *U* the optional units (must not contain special characters). For example, 10MHz represents 10^7 Hz.

```
... → num, ...
```

`«N[.M]»e«E[U]»`: a real number in scientific notation

The number is pushed on the stack into the *x* register. *N* is the integer portion of the mantissa and *M* is an optional fractional part. *E* is an integer exponent. *U* the optional units (must not contain special characters). For example, 2.2e-8F represents 22nF.

```
... → num, ...
```

`0x«N»`: a hexadecimal number

The number is pushed on the stack into the *x* register. *N* is an integer in base 16 (use a-f to represent digits greater than 9). For example, 0xFF represents the hexadecimal number FF or the decimal number 255.

```
... → num, ...
```

`0o«N»`: a number in octal

The number is pushed on the stack into the *x* register. *N* is an integer in base 8 (it must not contain the digits 8 or 9). For example, 0o77 represents the octal number 77 or the decimal number 63.

```
... → num, ...
```

`0b«N»`: a number in binary

The number is pushed on the stack into the *x* register. *N* is an integer in base 2 (it may contain only the digits 0 or 1). For example, 0b1111 represents the octal number 1111 or the decimal number 15.

```
... → num, ...
```

`'h«N»`: a number in Verilog hexadecimal notation

The number is pushed on the stack into the *x* register. *N* is an integer in base 16 (use a-f to represent digits greater than 9). For example, 'hFF represents the hexadecimal number FF or the decimal number 255.

```
... → num, ...
```

`'d«N»`: a number in Verilog decimal

The number is pushed on the stack into the *x* register. *N* is an integer in base 10. For example, 'd99 represents the decimal number 99.

```
... → num, ...
```

`'o«N»`: a number in Verilog octal

The number is pushed on the stack into the *x* register. *N* is an integer in base 8 (it must not contain the digits 8 or 9). For example, 'o77 represents the octal number 77 or the decimal number 63.

```
... → num, ...
```

'b«N»: a number in Verilog binary

The number is pushed on the stack into the *x* register. *N* is an integer in base 2 (it may contain only the digits 0 or 1). For example, 'b1111 represents the binary number 1111 or the decimal number 15.

```
... → num, ...
```

5.17.9 Number Formats

si[«N»]: use SI notation

Numbers are displayed with a fixed number of digits of precision and the SI scale factors are used to convey the exponent when possible. If an optional whole number *N* immediately follows *si*, the precision is set to *N* digits.

eng[«N»]: use engineering notation

Numbers are displayed with a fixed number of digits of precision and the exponent is given explicitly as an integer. If an optional whole number *N* immediately follows *sci*, the precision is set to *N* digits.

Engineering notation differs from scientific notation in that it allows 1, 2 or 3 digits to precede the decimal point in the mantissa and the exponent is always a multiple of 3.

sci[«N»]: use scientific notation

Numbers are displayed with a fixed number of digits of precision and the exponent is given explicitly as an integer. If an optional whole number *N* immediately follows *sci*, the precision is set to *N* digits.

Scientific notation differs from engineering notation in that it allows only 1 digit to precede the decimal point in the mantissa and the exponent is not constrained to be a multiple of 3.

fix[«N»]: use fixed notation

Numbers are displayed with a fixed number of digits to the right of the decimal point. If an optional whole number *N* immediately follows *fix*, the number of digits to the right of the decimal point is set to *N*.

hex[«N»]: use hexadecimal notation

Numbers are displayed in base 16 (a-f are used to represent digits greater than 9) with a fixed number of digits. If an optional whole number *N* immediately follows *hex*, the number of digits displayed is set to *N*.

oct[«N»]: use octal notation

Numbers are displayed in base 8 with a fixed number of digits. If an optional whole number *N* immediately follows *oct*, the number of digits displayed is set to *N*.

bin[«N»]: use binary notation

Numbers are displayed in base 2 with a fixed number of digits. If an optional whole number *N* immediately follows *bin*, the number of digits displayed is set to *N*.

vhex[«N»]: use Verilog hexadecimal notation

Numbers are displayed in base 16 in Verilog format (a-f are used to represent digits greater than 9) with a fixed number of digits. If an optional whole number *N* immediately follows *vhex*, the number of digits displayed is set to *N*.

vdec[«N»]: use Verilog decimal notation

Numbers are displayed in base 10 in Verilog format with a fixed number of digits. If an optional whole number *N* immediately follows *vdec*, the number of digits displayed is set to *N*.

`voct[«N»]`: use Verilog octal notation

Numbers are displayed in base 8 in Verilog format with a fixed number of digits. If an optional whole number *N* immediately follows *voct*, the number of digits displayed is set to *N*.

`vbin[«N»]`: use Verilog binary notation

Numbers are displayed in base 2 in Verilog format with a fixed number of digits. If an optional whole number *N* immediately follows *vbin*, the number of digits displayed is set to *N*.

5.17.10 Variable Commands

`=«name»`: store value into a variable

Store the value in the *x* register into a variable with the given name.

```
... → ...
```

`«name»`: recall value of a variable

Place the value of the variable with the given name into the *x* register.

```
... → value of «name», ...
```

`vars`: print variables

List all defined variables and their values.

5.17.11 Stack Commands

`swap`: swap x and y

The values in the *x* and *y* registers are swapped.

```
x, y, ... → y, x, ...
```

`dup`: duplicate *x*

The value in the *x* register is pushed onto the stack again.

```
x, ... → x, x, ...
```

alias: enter

`pop`: discard x

The value in the *x* register is pulled from the stack and discarded.

```
x, ... → ...
```

alias: clrx

`lastx`: recall previous value of x

The previous value of the *x* register is pushed onto the stack.

```
... → lastx, ...
```

`stack`: print stack

Print all the values stored on the stack.

`clstack`: clear stack

Remove all values from the stack.

```
... →
```

5.17.12 Miscellaneous Commands

`rand`: random number between 0 and 1

A number between 0 and 1 is chosen at random and its value is pushed on the stack into *x* register.

```
... → rand, ...
```

`` `«text»` ``: print text

Print "text" (the contents of the back-quotes) to the terminal. Generally used in scripts to report and annotate results. Any instances of $N or ${N} are replaced by the value of register N, where 0 represents the *x* register, 1 represents the *y* register, etc. Any instances of $Var or ${Var} are replaced by the value of the variable *Var*.

`"«units»"`: set the units of the x register

The units given are applied to the value in the *x* register. The actual value is unchanged.

```
x, ... → x "units", ...
```

`>«units»`: convert value to given units

The value in the *x* is popped from the stack, converted to the desired units, and pushed back on to the stack.

```
x, ... → x converted to specified units, ...
```

`(...)«name»`: a user-defined function or macro.

A function is defined with the name «name» where ... is a list of commands. When «name» is entered as a command, it is replaced by the list of commands.

`quit`: quit (:q or ^D also works)

alias: :q

`help`: print a summary of the available features

`?[«topic»]`: detailed help on a particular topic

A topic, in the form of a symbol or name, may follow the question mark, in which case a detailed description will be printed for that topic. If no topic is given, a list of available topics is listed.

`about`: print information about this calculator

5.18 Releases

5.18.1 Latest development release

Version: 1.10
Released: 2022-12-27

5.18.2 1.10 (2022-12-27)

- add caching for bitcoin prices

5.18.3 1.9 (2022-01-04)

- add *unit conversions*.
- add bitcoin quotes via *unit conversions*.
- allow Bitcoin Unicode characters (and ș) in units.
- rename *eng* to *si*.
- add new *eng* that uses exponential notation with exponent constrained to be a multiple of 3.
- drop support for Python 2.
- make more extensive use of Unicode.

5.18.4 1.8 (2021-11-10)

- nits

5.18.5 1.7 (2020-08-18)

- mag and ph now consume *x* register rather than duplicate it.
- implement *lastx*.
- loosen regular expression that matches numbers to allow scale factor to be optional.
- allow typical Unicode characters in (°, Å, , , Ω, and ℧) in units.
- add support for comments.
- genindex