The Double Calculator

- User’s Manual -

**--] Preface**

The purpose was creating a programmable engineering calculator for technical office and invoice, supporting automation, variables, mathematical functions, logic, file operations and multitasking.

The software comes in two different interfaces, console (Con) and GUI (Cute);

You may also download the latest version from <http://sites.google.com/site/trailofamadman>.

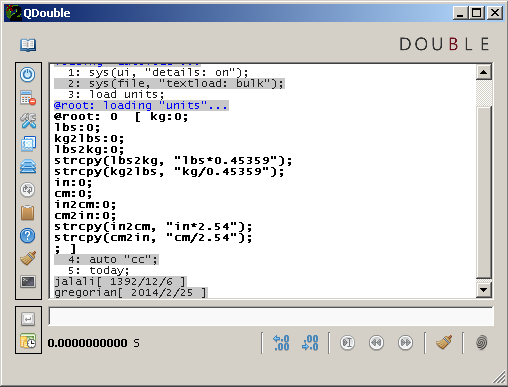


Fig 1. Cute schema

Enter expression in the input field, press [ Enter ] to see results; there is a virtual keypad exists in the Cute.

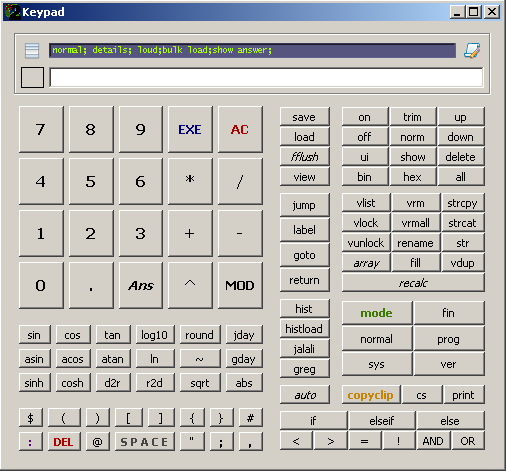


Fig 2. GUI keypad

Some Cute specific features are:

1. Calculation history access using [ Up ] and [ Down ].
2. Running expression by pressing [ Space Bar ] twice.
3. Variable, function and command name auto completion by pressing [ Tab ].
4. Erasing input field using [ Esc ].
5. Moving from output screen to input field using [ Tab ].
6. Programming window with [ F2 ].
7. Moving between output, programming and single line windows using [ Ctrl+Up ] and [ Ctrl+Down ].
8. Array editor [ aedit(*array*); ].
9. Graphical plot window [ plot(*array\_x, array\_y*); ].

Some Con specific features are:

1. Running argument as expression
2. Loading argument as file (double load *filename*).

Please note that the Cute version is easier to use for everyday calculations, but the Con is faster and more stable for sensitive use; the linux version is much more stable.

**--] Number Formatting:**

Decimal (Base 10): 123, -123, 123.456, -123.456

Hexadecimal (Base 16): 0x1b

Binary (Base 2): 0b101010

Scientific: 1.2e\+2, -1.2e\-2, 1.2e3

Date: 20130809 or “2013/8/9”

**--] Basic Mathematics:**

Basic operators are plus (+), minus/negative (-), multiply (\*), division (/), parenthesis (), array index ( [] ), bitwise or (|), bitwise and (&) and variable definition/assignment (=).

For an ordinary calculation, write expression ( e.g. 1+1 ) in the input field, then press [ Enter ], the input field will be cleared and the results will be shown in the output screen. The last answer is always accessible from the ‘ans’ system variable.

The expression will calculate regarding to the operator precedency, and the parenthesis could be used to change this behavior; For example, the answer of expression 1+1\*2 is 3, but (1+1)\*2 equals to 4.

The software includes some shortcuts for faster calculations for some basic operators like +,-,\*,/; For example a single +1 in the input field will increase the last answer by one ( this equals to ans+1 ), please note that expressions containing a negative as the first character (like -1) will act as minus, to use a negative value at first you may put it in parenthesis, e.g. (-1), thus a -1+2 equals to last\_answer-1+2 but a (-1)+2 equals to 1.

This program is also able to calculate multiple expressions in a single line, separated by a semicolon (‘;’), for example 1+1;+1; equals to 3.

Negative values could be shown by a ‘-‘ sign just before the value, for example 1+-1 equals to 0. This includes variables, for example 1--ans equals to 1-(-ans) and expressions like 1\*-1 are acceptable too.

The last operation is repeatable by pressing a single [ Enter ] on the empty input field, for example, type +1 and press [ Enter ] multiple times.

To avoid running an expression ( commenting out ), put an ‘#’ sign at the beginning, for example 1;#+2;+3; equals to 4.

**--] Logical operators**

These are greater than (>), less than (<), equals to (= or ==), greater than or equals to (>=), less than or equals to (<=), not equal to (!=), logical and (&&) and logical or (||); these operators are mostly used inside the logical expressions in conjuction with ‘if’ and ‘elseif’ functions; for example, if a equals to 1, b equals to 2 and c equals to 3, the result of expression

‘if(a>1 || b<c){1;} else {2;}’

Will be 1, please refer to the logical functions for more info.

**--] Special operators**

- operator ‘#’: used to comment out an expression.

- operator ‘@’: to access a variable in a background process and in the pattern @*parent*@*child*@*variable,* please note that a child process can not access it’s parent’s variables.

- operator ‘$’: to exit programming mode.

**--] variables**

Simple variables could be defined using ‘=’ operator; if not already defined, they will be created, else their value will change using this operator, but the definition will stay same in memory until changed using ‘strcpy’ function. This behavior in cunjuction with ‘recalc’ function makes variables much more flexible and behave like simple macros/functions, for example:

Assuming variables ‘a’ and ‘b’ are not defined to software:

a=1 – assigns value and definition of variable ‘a’ to ‘1’.

b=a+1 – defines variable ‘b’ as ‘a+1’ and sets it’s initial value to ‘2’.

a=2 – assigns value of ‘a’ to ‘2’, but the definition remains as ‘1’.

recalc(b) or re(b) will calculate value of ‘b’ again and sets it to ‘4’.

re(a) will reset the value of ‘a’ to ‘1’.

There is also another way to use variables with ‘recalc’, variables may be defined as blocks to be used as more complicated macros, define a variable as *variablename*={*expression*;} and re-calculate variable using ‘recalc’; for example, define a={b+1;print(ans);}, with each call to ‘recalc(a)’, the value of b+1 will be calculated and the results will print.

The program supports arrays and matrixes (2d arrays), definable using ‘array’ function and accessible using ‘[]’ operators; for example, expression ‘array(a[3])’ or a[3]=0; defines array variable ‘a’ with three elements and assigns all of them value ‘0’, accessible with a[0], a[1] and a[2]. Expression ‘array(a[3], 5, 6)’ repeats the previous operation but assigns value ‘5’ to a[0], ‘6’ to a[1] and ‘0’ to a[2], using = operator is another way to define and assign arrays, e.g. a[3]=1 defines array a with 3 members and assigns value 1 to a[2].

‘array(a[2][3])’ or a[2][3]=0; creates a 2\*3 (2 columns and 3 rows) matrix and assigns all members value ‘0’, the members will be indexed from a[0][0] to a[1][2]; ‘array(a[2][3], 1, 2, 3)’ does the same but assigns values ‘1’ to a[0][0], ‘2’ to a[0][1] and ‘3’ to a[1][0].

Using ‘array’ or = operator, array size could be expanded; in this case current values of the expanding array will remain untouched.

Variables could be protected to avoid accidental change, please refer to the section ‘variable control functions’ for more info.

Some variables are reserved inside the software, they are ‘ans’ to keep last answer, ‘clip’ to access clipboard contents, ‘pi’ and ‘e’.

**--] Mathematical functions**

Including trigonometric functions like ‘sin’, ‘cos’, ‘tan’, arc versions like ‘asin’ (adding an ‘a’ to the beginning), hyperbolic versions of above like ‘cosh’ (adding an ‘h’ to end); logarithmic functions like ‘log’ and ‘ln’; conversion functions like ‘radtodeg’ and ‘degtorad’; square functions like ‘sqrt’; absolute functions like ‘abs’, integer functions like ‘intval’; round functions like ‘round’ and ’~’ and date calculation functions like ‘gday’ and ‘jday’.

* Trigonometric functions:

1. sin: returns sines of the given degree number, e.g. sin(90) equals to 1.
2. cos: cosines.
3. tan: tangent.
4. asin: arc sines.
5. acos: arc cosines.
6. atan: arc tangent.
7. sinh: hyperbolic sines.
8. cosh: hyperbolic cosines.

* Logarithmic functions:

1. log or log10: logarithm of the given value in base 10. e.g. log(10) or log10(ans).
2. ln: natural logarithm (base e).

* Conversion functions:

1. radtodeg or r2d: converts a radian value to degree. e.g. r2d(pi) equals to 180, and cos(r2d(pi)) equals to -1.
2. degtorad or d2r: converts degrees to radians.

* Square root functions:

1. sqr or sqrt: calculates the second root of a value.

* Absolute value functions:

1. abs: returns absolute value of the given value.

* Round functions:

1. round: used to round a number, the syntax is

round(*number, precision, mode*); first and second arguments are the number and needed precision, the mode argument could be ‘up’ to round up, ‘down’ or ‘dn’ to round down, ‘tr’ or ‘trim’ to round trim, and ‘nr’ or ‘normal’ for a normal round i.e. increase last digit by 1 if the last+1 decimal digit is greater than or equal to ‘5’, else decrease it.

This function could be used with one or two arguments, the default mode argument is ‘normal’, and the default precision is ‘2’. For example, round(90.12, 1, tr) returns 90.1, round(90.125, 2) and round (90.125) both return 90.13.

1. ~: equals round(*value*, 2, normal) or round(*value*).

* Date calculation functions:

1. jday: returns day number of a Jalali (Shamsi) calendar date, this function could be used to calculate the days between two dates. e.g. jday(13920809)-jday(“1392/8/1”)
2. gday: same as above for Gregorian calendar, please note that using ‘gday’ and ‘jday’ together in an expression will not return correct results because of the different Gregorian and Jalali calendar day numbers, it is recommended to convert dates to a single system first, see ‘greg’ and ‘jalali’ date conversion functions for more info.

**--] Scripting functions**

Including logical decision functions ‘if’, ‘elseif’ and ‘else’; clipboard functions ‘copyclip’ and ‘copyclipsep’; variable control functions ‘vlock’, ‘vunlock’, ‘vstr’, ‘vlist’, ‘vexp’, ‘vdesc’, ‘vrmall’, ‘vrm’, ‘recalc’, ‘afill’, ‘array’; file functions ‘load’, ‘save’, ‘fflush’ and ‘view’; flow control functions ‘label’, ‘goto’, ‘return’ and ‘jump’; automation function ‘autorun’, history functions ‘history’ and ‘histload’ and date system conversion functions ‘greg’ and ‘jalali’.

Please note that, these functions should be used in separated expression in order to work correctly; e.g. vrm(a); vlist;

Some functions take arguments as a list, the list item separation character is ‘,’; e.g. vrm(a, b, c);

* logical decision functions:

1. if: the syntax is ‘if(*logical\_expression*){*expression\_to\_run\_if\_true*;}’, examples of logical expressions are, 1<2 (true) or 1\*1<2&&3>4 (false). If the logical expression is true, the expression inside the curly brackets will run, else, if there is an ‘elseif’ function following, the logical expression of it (‘elseif’) will be checked, the rest is just like ‘if’. If non of ‘if’ or ‘elseif’ expressions are true and there exists an ‘else’ function, the expression inside the curly brackets of it (‘else’) will run.

e.g. if(a<2&&b>2) {1;a:3;} elseif (c=a|b!=3) {0;+1;} else {+2;}

1. elseif: see ‘if’.
2. else: the syntax is ‘else {*expression*;}’, has no logical expression; please note that the space between ‘else’ and ‘{*expression*;}’, this is necessary!

* Clipboard functions:

1. copyclip or cc: this function used to copy values to the system clipboard, the syntax is ‘cc(*expression*)’; expression could be a value, string, variable or a complete mathematical expression; for example if variable ‘hello’ is defined as ‘1000’, the call cc(hello) will copy value 1000 to the clipboard, but cc(“hello”) copies word ‘hello’ to the clipboard, running ‘cc’ with no arguments will copy the value of system variable ‘ans’ to the clipboard, and running cc(fin) will copy the calculation in financial format to the clipboard in ‘financial mode’.
2. copyclipsep or cs: this works just like ‘cc’ except that it formats numbers in thousands separated form.
3. num2text or n2t: converts values from numerical to text and copies results to the system clipboard, syntax is num2text(*value*); running this function without argument converts the value of *ans* variable.

* Variable control functions:

1. recalc or re: this function re-calculates the variable’s formula and sets the results to it’s value. The syntax is re(variable\_list…) or re; running this function without arguments recalculates all non-locked variables defined in memory. Passing an array name as argument makes function recalculating row *i* of an array in the form of re(*array\_name[i]*) or column *j* of the given array in the form of re(*array\_name[-1][j]*) or all members of an array in the form re(*array\_name*).
2. vrm: this function removes non-locked variables or arrays from memory, the syntax is vrm(*variable\_list*…) or vrm all; the *all* argument removes all non-locked variables in memory.
3. vdesc: this function used to add, remove, change and show the description of a variable. The syntax is vdesc(*variable, “description”/command*), first argument is variable name, second argument is optional and can be on of the *remove, show* or *“description string”* values, the default value for the second argument is *show*.
4. vexp or strcpy: this function used to assign expressions to a variable, the syntax is strcpy(*variable*, “*expression*”); e.g. strcpy(var1, str(var2));
5. vcat or strcat: this function used to combine expressions of two variables, use like strcpy.
6. vlist: this function lists defined variables, syntax is vlist(*variable*/*array\_name*) or vlist; the variable name argument accepts common wildcards (\* all, ? any) inside double cotations ( “ ), e.g. vlist(“\**part\_of\_variable\_name*??”);
7. vstr or str: returns variable expression, syntax is str(*variable*);
8. vstrcmp or strcmp: compares expression strings of two variables, or strings; syntax is strcmp(*variable/string, variable/string*); e.g. if(strcmp(var, “+1”)){print(“not plus one!”);}
9. vlock or vlk: locks variables, shows locked variable list if used without arguments; syntax is vlock(*variable\_list…*) or vlock;
10. vunlock or vun: unlocks variables, use like vlock.
11. vren or rename: renames variables; syntax is rename(*old\_variable\_name, new\_variable\_name*);
12. vdup or copyvar: duplicates variable, syntax is vdup(*destination\_variable\_name, source\_variable\_name*); this function also copies arrays and matrixes fully or partially when used like vdup(*dst*, *src*[*i*]) to copy one row or vdup(*dst*, *src*[-1][*j*]) to copy one column, where i is row number and j is column number.
13. array: defines arrays and matrixes, syntax is array(*array\_name[size], value\_list…*) or array(*matrix\_name[row][column], value\_list…*); the first argument is the name of the array and the number inside the brackets is the size of the array, the index starts from 0 and ends to size-1. E.g. if variable i equals to 5, the function call array(a[i], 1, 2) creates array a with 5 members indexing from 0 to 4, assigns 1 to a[0], 2 to a[1] and 0 to others. To access members, expressions like a[1+1], a[*variable*] and a[+2] are valid; calling array with an existing array name and size bigger than current array size, expands the array.
14. fill: fills array with an specific value, syntax is fill(*array\_name*, *value*) or fill(*array\_name*, *expression*); to assign a value to all rows of a matrix use fill(*array\_name[row], value*) and to assign to a column use fill(*array\_name[-1][column], value*).

* File functions:

1. load: loads an script file, script files are text files written in double’s language. Syntax is load(“*filename*”), lines beginning with a # are commented out.
2. save: used to save expressions into a text file, syntax is save(“*filename*”/*command*); if file does not exist already, it will be created, else the expressions will be added to the end of file. The commands are *stop* to pause saving and *resume* to resume saving.
3. fflush: writes save buffer to the file in the run-time, else the changes will be saved on the disk on exit, an auto “fflush;” could be used to write every change at once, this function takes no arguments.
4. view: used to view file contents, the syntax is just like load.

* Flow control functions:

1. label: used to create a label to be refered by goto, syntax is label *label\_name*; e.g. expression i:0;label a;i:i+1;if(i<3){goto a}; increases i till i < 3;
2. goto: jumps back to label defined by label function, syntax is goto *label\_name*; there is jump forward, the label must be already defined.
3. return: this function exits from a running code, this function takes no arguments, e.g. if(i<3){return;} will breaks the running code if variable i is less than 3.
4. jump: ignores next expressions as passed argument, syntax is jump(*value*); no arguments means jump(1). E.g. 1;jump(2);+1;+1;+1; equals to 2;
5. fork: this function calculates an expression on the background; thus, the user will be able to continue normal operation on time consuming operations, forked process variables are accessible via @ operator. The syntax is fork(*fork\_name*, “*expression*”), fork(*fork\_name*, *variable*), and fork; running this function with no arguments will display a list of background processes.
6. kill: used to kill background processes, syntax is kill(*fork\_name*) or kill all; the calculation must have been finished already in order to kill process; if not, try break.
7. switch: switches between background processes, syntax is switch(*fork\_name*), switch parent and switch root. Running with no arguments will display current process. the calculation must have been finished already in order to switch process; if not, try break.
8. break: this function breaks current running background process, syntax is break(*fork\_name*) or break all.
9. sleep: delays running current process for n microseconds, syntax is sleep(*n*).
10. exec: runs a shell command and displays results, syntax is exec(*argument\_list…*).
11. autorun or auto: this function runs the given expression after finishing code piece, syntax is auto(*“expression”/command/variable*), auto off, auto on and auto; auto off, pauses the autorun and auto on resumes, running with no arguments displays the current autorun expression. E.g. auto(“+1;cc;”), adds last answer by one and copies results into the clipboard after running each expression, passing a variable as argument will autorun the variable’s expression.

* History functions:

1. history or hist: shows calculation history.
2. histload: loads and runs the expression n, syntax is histload(*n*);

* Date conversion functions:

1. jalali or shamsi: converts Jalali date to Gregorian depending on the locale, syntax is jalali(*numericdate*) or jalali(“*date*”); e.g. jalali(13921208) or jalali(“1392/12/8”);
2. greg or miladi: converts Gregorian date to Jalali, syntax is like jalali.
3. today: returns today’s date depending on the locale, syntax is today;
4. jdate: converts Jalali day number to date depending on the locale, syntax is jdate(*dayno*); see jday.
5. gdate: converts Gregorian day number to date, see jdate.

* Control functions:

1. mode: controls interaction mode, has three states, *normal* or *nr* for normal operation mode, *fin* for finance and *prog* for programming operation mode. Syntax is mode(*operation\_mode*); in the programming mode, each line will not run after pressing [ Enter ] key, to run the program use a single ; in an empty line and press [ Enter ], please note that the end of line must be terminated by a ; and should be done manually by the user, to exit this mode use a $ sign.

* Display functions:

1. print or pr: this function prints arguments to the output, syntax is print(*argument\_list…*), arguments could be variables, values, strings surrounded by ‘ or “, reserved values and expressions, e.g. print(“answer is”, “ “, ans); prints the string “answer is” following by value of ans variable.
2. tr or trim: insert line break.
3. hex: next value in hexadecimal.
4. bin: next value in binary.
5. all: more details on next argument, try on arrays.
6. plot: this function draws values of two arrays, for example if x and y are defined as two arrays, plot(x, y) draws values of them on a screen.
7. aedit: array editor (platform depended), use aedit(x) or without argument.

**--] Commands:**

1- sys: tunes software parameters, see section sys.

1. ver: returns engine version, useful for compatibility checks.
2. help: online quick help.
3. exit: exits software.

**--] Usage notes:**

1- the software loads file ‘autoload’ automatically on the startup.

1. To make loops try expressions like:

0; label a;+1; if(ans == 10000){jump;}; goto a;

Using expressions like:

0; label a;+1; if(ans < 10000){goto a;};

Consume more resources and lower stability, the first expression runs the expression inside the brackets once, on the other hand the second one runs it 10000 times to achieve the same results!

1. If load function does not load some big files completely, tune expression buffer via sys, see section sys for more info.

**--] Sys:**

1- sys(ui, “hilight: on/off/stat”): shadowed output control.

1. sys(ui, “groupnum/numsep: on/off/stat”): thousands separator control.
2. sys(ui, “decimal|decim: trim/fix/stat/n/+/-”): number precision control.
3. sys(ui, “verbose|v: off/on/stat”): controls verbosity.
4. sys(ui, “answer|ans: off/on/stat”): answer display control.
5. sys(ui, “details|det: off/on/stat”): details display control.
6. sys(ui, “locale: fa/en/stat”): locale control.
7. sys(ui, “nerd: off/on”): nerd info!
8. sys(file, “textload: line/bulk/stat”): text file loading mode (use bulk)
9. sys(buf, “exp/tok/out/hist: stat/n/+/-“): buffer size control (in bytes)

--] Info:

This document intended to be used on:

Engine Version 0.92.ddlus-B

UI Version Cute\_1G & Con\_0.1

By:

**Babak Akhondi <** [**jaysi13@gmail.com**](mailto:jaysi13@gmail.com) **> summer 2014**