# Halcyon Calc Documentation

# RPN Quick Start

Most people are familiar with standard arithmetic notation which looks something like this:

1 + 3 + 6

In this example, 1 is added to 3 to give 4 which is then added to 6 which results in the answer 10. Things get a bit more complicated when different operations are included:

1 + 3 * 6

Without a standard order of operations, this expression could be interpreted in two different ways. You could evaluate "1 + 3" first to give 4 which is multiplied by 6 to result in 24. Or, you could evaluate "3 * 6" first to give 18 which is then added to 1 to result in 19. Depending on what you start with, you get different answers. For this reason, multiplication is always performed before addition (or subtraction) so that is the standard order of operations in mathematics. So, 19 is the right answer here.

To make things explicit, we can use brackets:

(1 + 3) * 6

In this case, 1 + 3 is evaluated first because it is in brackets so the correct answer is 24. This should all be familiar from elementary and high school mathematics classes. But, most people have not encountered reverse Polish notation.

Here is an example:

1 3 + 6 +

This is the equivalent version of "1 + 3 + 6" in RPN. In order to understand this notation, you must understand the concept of a stack of numbers. There are two basic operations with a stack. You can push a number onto the bottom of the stack or you can pop the most recently pushed number off of the stack. This may sound very limiting but with these two basic operations, you can do quite a bit of manipulation.

Now to read "1 3 + 6 +", you get:

1. The number 1 is pushed onto the stack.
2. The number 3 is pushed onto the stack so that 3 is now at the bottom of the stack and 1 is next to the bottom.
3. The operation + is executed which pops two numbers off the stack, adds them together and pushes the result back onto the stack. In this case, 3 and then 1 are popped off the stack, they are added together to get 4 and then 4 is pushed onto the stack. At this point 4 is at the bottom of the stack.
4. The number 6 is pushed onto the stack so that 6 is now at the bottom of the stack and 4 is next to the bottom.
5. The operation + is executed which pops 6 and 4 off the stack, evaluates 4 + 6 to get 10 and pushes 10 into the stack.

Because 10 is on the stack at the end of evaluating the expression, 10 is the answer which is

consistent with "1 + 3 + 6".

Now let's look at "1 + 3 * 6". In this case, this would be written as the following in RPN:

1 3 6 * +

To see how this works, we can read through it like this:

1. The number 1 is pushed onto the stack.
2. The number 3 is pushed onto the stack so that 3 is now at the bottom of the stack and 1 is next to the bottom.
3. The number 6 is pushed onto the stack so that 6 is now at the bottom of the stack, 3 is next to the bottom and 1 is above that.
4. The operation * is executed which pops 6 and 3 off the stack, evaluates 3 * 6 to get 18 and pushes 18 onto the stack. So, 18 is now at the bottom of the stack and 1 is next to the bottom.
5. The operation + is executed which pops 18 and 1 off the stack, evaluates 1 + 18 to get 19 and pushes 19 onto the stack.

At the end of this, 19 is left on the stack which is consistent with the answer expected for "1 + 3 * 6". The other interesting example is "(1 + 3) * 6" from above. In RPN, that looks like:

1 3 + 6 *

Note that brackets are not required in RPN to get the order of operations you want. This is how it gets evaluated:

1. The number 1 is pushed onto the stack.
2. The number 3 is pushed onto the stack so that 3 is now at the bottom of the stack and 1 is next to the bottom.
3. The operation + is executed which pops two numbers off the stack, adds them together and pushes the result back onto the stack. In this case, 3 and then 1 are popped off the stack, they are added together to get 4 and then 4 is pushed onto the stack. At this point 4 is at the bottom of the stack.
4. The number 6 is pushed onto the stack so that 6 is now at the bottom of the stack and 4 is next to the bottom.
5. The operation * is executed which pops 6 and 4 off the stack, evaluates 4 * 6 to get 24 and pushes 24 into the stack.

As before, this evaluates to the correct answer. In all of the examples so far, the only operations we have seen are addition and multiplication. But with subtraction and division, the order of the numbers is very important. The expression "3 - 1" is very different from "1 - 3". In RPN, the order is the same as you are used to. So, "3 - 1" in standard notation is "3 1 -" in RPN.

Also, all of the examples seen so far are "binary" operations meaning they take two numbers and return one number. For example, an addition takes two numbers, adds them and then produces a result. There are also unary operations which take a single number, performs some operation on that number and produces a result. An example of this are the trig functions like sin. So, the expression "sin(5)" in standard notation would look like "5 sin" in RPN. A unary operation like sin in RPN pops a single number off the stack, performs its operation (ie calculates the sin of 5) and the pushes the result onto the stack. This can then be used to evaluate something more complicated like "sin(5 - 3)" which would look like "5 3 - sin" in RPN.

At this point, you may be getting more comfortable reading something in RPN, but how do you take a standard notation expression and convert it into RPN. It is actually quite easy. In order to understand how to do this, let's convert "3 * 8 - (2 + 3)" into RPN.

First, read the expression left to right. The first thing we see is the number 3 so we should push 3 into the stack. At this point, the RPN version of this expression looks like "3". The next thing we see is the multiply operation. But in RPN, we need to skip past this operation for now and find the other number which is being multiplied. So, we look at the number 8 and we push that onto the stack. The RPN expression looks like "3 8".

The next thing is the subtract operation. Because of the order of operations, the multiply should be evaluated before the subtract, so we need to perform the multiply at this point. The RPN expression now looks like "3 8 *". We have seen the subtract operation but we need to find the other number we are using in the subtract. So, we continue before we do the subtract.

The next thing we see is the open bracket, which we ignore for now. After that is the 2 so we push that onto the stack resulting in an RPN expression which looks like "3 8 * 2". We see a addition operation next. The question we need to ask at this point is should the subtract or the addition be performed first. Because of the brackets, the addition should be performed first. So, we need to get the other number being added which follows the + sign and is a 3. We push 3 onto the stack resulting in an RPN expression which looks like "3 8 * 2 3 +".

The final thing we need to do is the last operation which is the subtract. The final RPN expression is:

3 8 * 2 3 + -

RPN may seem unnatural at first but most people will very quickly figure out how to work with it. The benefit of RPN over standard notation for a calculator is the stack. The stack represents a scratch pad which can grow or shrink as you need it. And you don't need to use brackets to try and get the expression you want. Once you are comfortable, you will be able to very quickly look at a formula in standard notation and enter it in RPN.

To put this in the context of Halcyon Calc, the last RPN expression above can be calculated in Halcyon Calc by pressing these buttons:

3 Enter 8 Enter * 2 Enter 3 Enter + -

The "Enter" key is used to push a number onto the stack. That takes 11 key-presses. In order to save key-presses, you don't need to press Enter when you execute an operation. Regardless of whether the operation is a binary or unary operation, the number which you have typed but not yet pushed is automatically push onto the stack before the operation is executed. So, you can simplify the above by pressing:

3 Enter 8 * 2 Enter 3 + -

This brings the count down to 9 key-presses, saving 2 Enters. But, either works of course.

The rest of the documentation for Halcyon Calc assumes familiarity with RPN so you may want to experiment with the calculator a bit. Try to calculate some specific equations. Start with some simple ones and progress to something more complicated, but still just using the standard add, subtract, multiply and divide operations which are clearly visible on the virtual keypad. Many

more operations are available but how to access and use them will be described elsewhere. Once you feel comfortable with RPN, dive into the rest of the docs to explore the real power of Halcyon Calc.

# Halcyon Calc - Working With The UI

This document describes the interface features of the calculator. When you launch the calculator for the first time, you will be presented with a UI which resembles the image below. Actually, in the picture, there are items on the stack and something being entered which you will not have on initial startup. Instead, the entry and the stack will be empty but for demonstrative purposes, it is useful to see some data in those areas of the UI also.

This image has labels on four key parts of the UI. Those elements are:

1. The Main Buttons
2. The Menu Buttons
3. The Command Line
4. The Annunciators
5. The Stack


Each of those elements of the UI are described below.

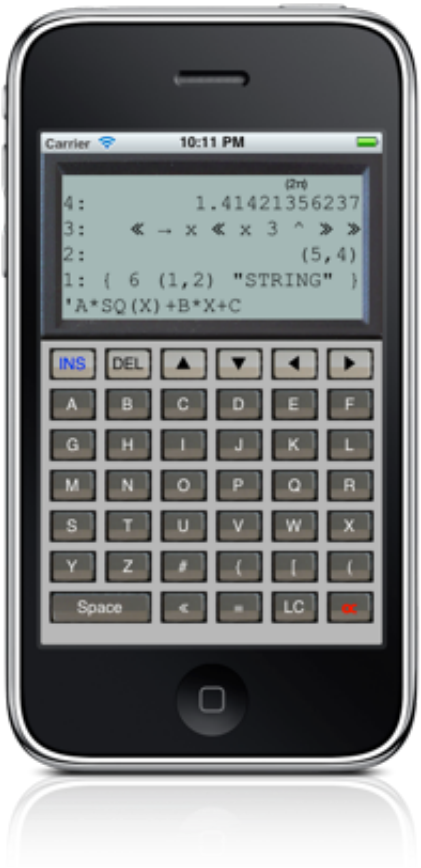
## The Main Buttons:

What is a calculator without buttons. On the image shown, you can see the standard buttons for the numbers 0 through 9. Also, there are the basic addition, subtraction, multiplication and division buttons. At the bottom, you will also see buttons for a period and a comma. The meaning of these buttons by default depends on "Region Format" setting on your device. In some regions (like much of Europe), the comma is used as the radix point separating the integer from the fractional part of a real number. In other regions (like much of North America), the period is used as the radix point. You can also use the RDX, operation to change this default behaviour to what you would like. Regardless, one of these buttons is used as a radix point, while the other is used as a separator. Peek at #4 in the image and you will see the complex number (5,4) and a real number 1.41421356237. So, in this image, period is the radix point and comma is the separator which is used, in this case, to separate the real and imaginary components of a complex number.

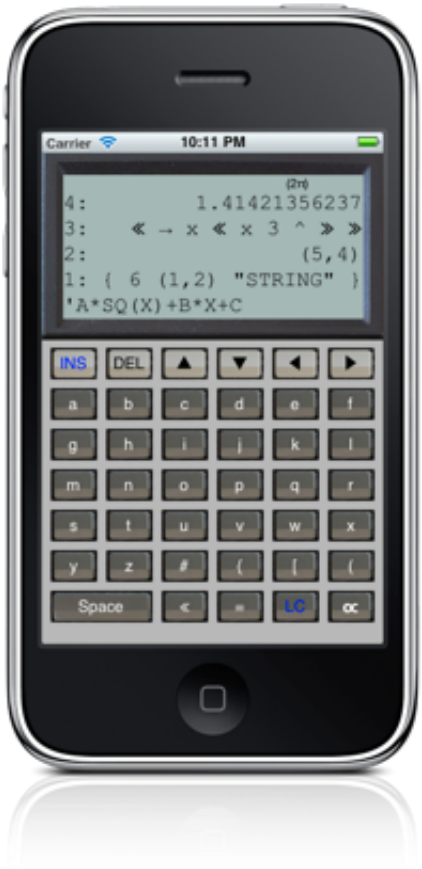
Other than these immediately recognizable buttons common on most calculator, there are others like "CHS" and "Drop" which will be explained in more detail later.

But before pressing any of the buttons in this area, try swiping to the left with your finger. And then swipe back to the right again.

For basic usage, you shouldn't require the buttons on the left page but those buttons give you access to features like lists, integer arithmetic, and symbol expressions.



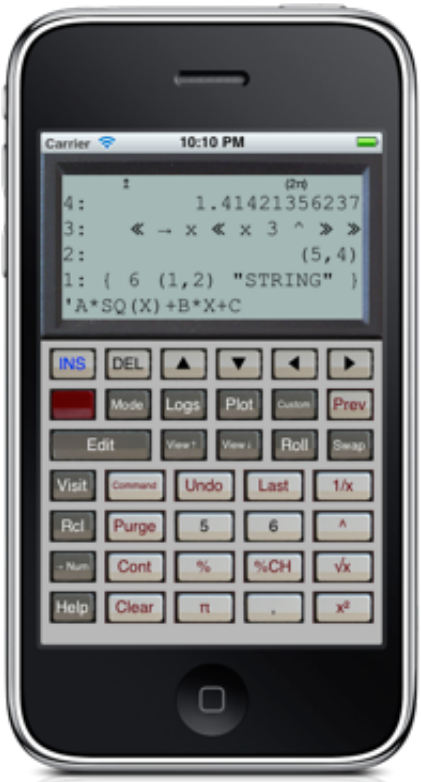

Also, if you tap the "LC" button on the left page, you can switch to lower case character mode. When you do so, the labels of the buttons affected automatically change to reflect that lower case mode is enabled. Also, the "LC" button glows blue which indicates that the calculator is in lower case mode. Blue text on a button is used to indicate that a corresponding mode is enabled and you will see that in other situations.



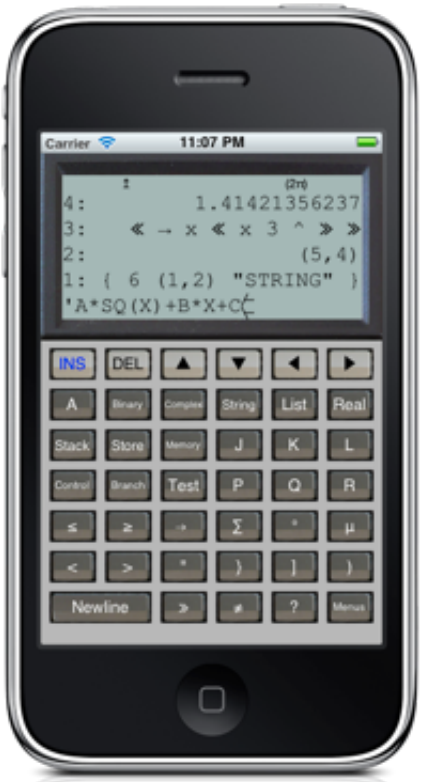

Lower case mode is "sticky". That means that when in lower case mode, the calculator will remain in that mode until you press the "LC" button again. You can even quit the calculator and return to it again later and it will still be in lower case mode.

On the right page of buttons, at the top left you will see a red button without a label. This is the "red shift" button and when pressed it gives you access to more capabilities, both on the left and the right page of buttons.

When you press red shift, the label on most buttons will change. However, some buttons have no other operation associated with them and the label remains the same after the red shift. The image to the left shows the right page of buttons with red shift on and note that 1, 5 and 6 are unchanged in the image. Lighter coloured buttons have their label drawn in red to indicate that red shift is on. Darker coloured buttons still are labelled in white to preserve readability.




If you press the red shift button a second time, the red shift turns off and the buttons return to their usual state. Also, red shift is not sticky. That means that red shift mode is automatically disabled after pressing any button.




If you swipe to the left while red shift is on, you will see that some of the buttons on the left have different operations also. Many of the button towards the top of the page have been replaced by menu buttons like "Binary" and "Stack". For some people, they may very rarely use the letter buttons but might want quick access to these menu buttons. At the bottom right of the page, there is a "▪Menu" button (when a button is described in red text proceeded with a red box, that button is accessible with a red shift) which swaps the behaviour of these buttons. When Menu mode is enabled, these buttons show their menu buttons when red shift is off and switch back to their letters when red shift is on. So, the "B" button will instead show the label "Binary" and open the binary menu when red shift is off but when red shift is on, it will become the "B" button. Depending on what you use more frequently, you may want to have menu mode on or off.

Also, menu mode is sticky. It stays on until you press the Menu button again. If you enable menu mode, the "▪Menu" button's label will be coloured blue to indicate it is on. Menu mode is also preserved if you quit and restart the calculator.

Finally, the buttons themselves can be pressed. If you press and release the button quickly, you may or may not get any visual feedback but an audible click, like from the iPhone keyboard will be heard (actually, on the original iPod Touch, you will not hear that click unless you use headphones). If you press and hold a button, the button will grow larger so you can get some feedback which button you have pressed. If that is the button you want, you can release your finger to activate that button. If it is not the button you were aiming for, move your finger outside the bounds of the button and release. When you move your finger, a different button will not be pressed but the button you did press will not activate.

## The Menu Buttons:

The menu buttons are a set of six dynamic buttons which change their label and behaviour depending on what menu has been selected. By default, the "INS", "DEL" and four arrow keys are shown on the menu buttons. The behaviour of these buttons will be described in the command line section. At any time, you can return to these six buttons by pressing the "❖" button which can be found on the right page, just to the right of the red shift button. The "❖" button is supposed to represent the four arrow keys on a single button which might help you remember what it is for.

The menu buttons themselves operate just like the main buttons. You get audible feedback when you activate the button and if you press and hold, you will get visual feedback when the button grows larger.

The default arrow buttons only have six menu buttons so a single page of menu buttons is all there are. But, if you press the "Trig" button on the right page, just to the right of the "❖" button a new set of buttons will slide in. The old buttons will slide up and the in this case, six trigonometry buttons will be visible. However, there are three pages of trig buttons. Swipe your finger towards the left on the trig menu buttons and a second set of trig buttons should be revealed. Swipe again to the left and you will see yet more buttons. If you try to swipe a third time, the buttons will slide to the left and then back again to indicate the end of the pages. Similarly, you can swipe to the right to see the previous page and swipe again to see the original page.

You can also use the "Next" and "▪Prev" buttons to move to the next or previous page of menu buttons. If you press "Next" when on the last page of buttons, you are returned to the first page. Similarly, if you press "▪Prev" when on the first page of buttons, you are sent to the last page of buttons.

The actual buttons which are available in these menus are best described in the Operations By Menu Reference.

## The Command Line:

In the upper part of the calculator is the display. The display contains the stack and it also can show the command line. When you launch the calculator for the first time, the command line will be empty. When the command line is empty, it will not be shown at all. Instead, the display will show up to four items on the stack, filling the display.

However, if you press the "1" button, that will insert a 1 onto the command line. The display will move the stack up and the command line will be visible. The command line is left justified so you will see that "1" character on the bottom left of the display. If you now press the "Enter" key, you

are instructing the calculator to interpret the contents of the command line and act on it. In this case, the calculator sees the number 1 so it puts that number at the top of the stack (which is actually at the bottom of the display, labelled as "1:", see the next section for details). At this point, the command line is cleared and your number 1 is on the stack.

Now try pressing the "2" button. The display again changes, showing the command line you have started to enter with that number at the bottom of the display. The stack moves up in the display to make room for the command line. Instead of pressing "Enter" again, press the "+" key. By default, most keys which are associated with an operation (like addition in this case) will cause the current command line, if any, to be interpreted before the operation executes. So, the "2" which is on the command line is interpreted by the calculator and because it is a number, it is pushed onto the stack. Then, the addition operation executes which gets the 1 and the 2 from the stack, adds them together and pushes the result, 3, onto the stack.

But, operations can also act differently sometimes. This time press the single quote button below the "Enter" key. This starts an expression. Then press the "1", then the "+" and finally the "2" buttons. Notice how this time, pressing "+" did not cause the calculator to interpret the command line and execute an addition. When you pressed the single quote and started building an expression, the calculator when into "algebraic" mode. In this mode, all operations which are valid in an expression will not execute right away but will instead be added to the command line. In this case, a "+" character was added to the command line so it should now look like:

```
'1+2
```

Similarly, if you press the SIN key from the Trig menu, then "SIN(" will be added to the command line. An operation which takes its parameters in brackets will add the open bracket automatically in algebraic mode. Look at the bottom right of the left page of main buttons and you will see the "∝" (alpha) button. With your expression still on the command line, you should see that the alpha button has its label drawn in red. That indicates that the command line is in algebraic mode. If you press the button, it will cycle between white, blue and red. White is normal mode. In that mode, pressing "+" or any other button associated with an operation will execute that operation. Red is algebraic mode, described here. Blue is alpha mode and is described more below. When you start an expression, the command line automatically goes into algebraic mode to make it easy to build that expression. But, you can override that mode at any time by pressing the "∝" button.

With your expression for 1+2 still on the command line, press "Enter". This causes the calculator to interpret the command line. It sees a valid expression and pushes that expression onto the stack. Note that it didn't calculate the result. See the Working With Expressions guide for more information about expressions.

The other mode controlled by the "∝" button is alpha mode. You can go into this mode manually by pressing the "∝" button until it is blue. Or, you can press the "▪" button (found on the left page) or the "≪" button (also found on the left page) to automatically go into the alpha mode. The double quote button indicates that you are entering a string (see the Working With Strings guide for more information). The "≪" character is used to being a program. By default, the calculator switches to alpha mode when entering a program.

In alpha mode, all buttons associated with operations will insert a character representation of themselves onto the command line. Unlike in algebraic mode, even operations which are not valid in an expression will be added to the command line. Also, brackets are never appended like

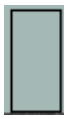
they are in algebraic mode. But, a space may be inserted before the operation to separate it from anything else already entered. A space is always added after the operation name. So, pressing the SIN button from the "Trig" menu in alpha mode will add " SIN " or "SIN " to the command line (a space is prepended to separate it from existing characters on the command line, if necessary).

Also, the command line presents another way to execute an operation. You can execute the SIN button from the "Trig" menu by pressing the button but you could also press the "S", "I", "N" and finally "Enter" buttons. If you know the name of the operation you want to execute, it may be quicker in some situations to type the operation rather than bring up the menu and hit the button.

No one is perfect and thankfully, the button "←" can be used to delete the character before the cursor on the command line. You can delete one character or, by pressing the button multiple times, you can delete many characters. If you delete all characters on the command line, the command line itself will disappear making more room on the display for the stack. But, that is not the only editing tools the calculator gives you for the command line.

If entering a great deal of information, the command line will wrap. If you press the "❖" button, the menu buttons will become editing buttons for the command line. The four arrow buttons allow you to move the cursor position on the command line. The INS button indicates whether the calculator is in insert mode or not. The INS button will be drawn with a blue label if insert mode is enabled. When in insert mode, any characters entered will be inserted into the command line at the cursor point. Characters after the cursor point are preserved. When not in insert mode, characters entered will overwrite any characters after the cursor point. The DEL button removes the character after the cursor. These buttons allow you to manipulate the command line and fix any mistakes you might have made.

You may have noticed that the cursor changes shape sometimes. There are two factors which influence the appearance of the cursor. First, the cursor appears different if the command line is in insert mode versus replace mode, as selected by using the INS button. Second, the cursor appears different based on the entry mode. This gives you an indication whether the calculator is in immediate, algebraic or alpha entry mode. The following table describes the different cursor appearances:

| | Insert Mode | Replace Mode |
|---|---|---|
| **Immediate Mode** |  |  |
| **Algebraic Mode** |  |  |
| **Alpha Mode** |  |  |

Sometimes, it is easier to start with a new command line than to edit the existing one. You can clear the entire command line by pressing the "Attn" button at the bottom left of the right page of buttons. The "Attn" button is short for attention and is a general "cancel everything" button. It can be used to clear a command line and also to interrupt a long running operation.

There are a couple more buttons which are useful for manipulating the command line. The "▪Edit" button allows you to put the item from the top of the stack onto the command line. From there, you can edit the command line and then when you press enter, the new value entered will replace the one you were editing on the top of the stack. Similarly, the "▪Visit" button allows you to do the same thing, but with any item on the stack. Push the number of the item on the stack you would like to edit and the press the "▪Visit" button. The item will now be on the command line and when you finish editing it, you can then replace that item on the stack with your new value.

Finally, the "▪Command" button allows you to recall one of the last four command lines entered on the calculator. Press the "▪Command" button once to recall the last command line. Press it a second time to recall the previous command line. Continue pressing it to recall the third and fourth command line. Press it a fifth time and it will return to the most recent command line. Once you have recalled the command line you are looking for, you can use the editing buttons to manipulate the command line before you enter it. Also, you can always press "Attn" to clear what you are doing if things go wrong for any reason.

The other guides will describe how to enter real numbers, complex numbers, lists etc. But, there are some tricks you can use with the command line which applies to all types of data you enter. The command line can be used to enter multiple items at a time. If you hit "1", then the space button and then "2" and finally "Enter", you will see that both 1 and 2 are pushed onto the stack. You can enter multiple things on a single command line if they are separated by a space, a new line or the "separator character". As described above, if the radix character is period, then the separator character is a comma or vice versa. So, assuming that the radix character is a period, you could also enter "1,2" into the calculator and both 1 and 2 will be pushed onto the stack. This can sometimes be a great short cut. If you are entering symbols like X and Y, you can put multiple symbols into the command line and keep the main buttons on the left page if you separate them with a space character.

If you double tap on the command line, options to copy and paste from/to the command line will appear. If you select copy, the current command line is stored onto the clipboard. You can then use that data in another application or paste that back into the calculator. Note that if you double tap on an item in the stack, you can copy that item onto the clipboard. Also, if you select paste after double tapping on an item on the stack, it actually pastes onto the command line, not onto that item in the stack. All data entry is to the command line.

## The Annunciators:

Along the top of the display, the calculator may show a series of icons which can help you identify what state the calculator is in. The following icons may be visible in this section:

| Annunciator | Indicates... |
|---|---|
| ◦ | There is a suspended program (see the Working With Programs guide for more information). |
| ↥ | The red shift key, "▪" has been pressed. |
| α | The calculator is in alpha entry mode. |
| ((●)) | The calculator is busy (press "Attn" to interrupt the calculator). |
| (2π) | The calculator is in radians mode. See RAD and DEG for more information. |

## The Stack:

The stack consists of a series of items which are results of computations, intermediate results or anything else which can be represented on the calculator (like lists or strings). The item at the top of the stack is at the bottom and is labelled "1:". In the picture, that item is the complex number (5,4). In the picture, there are two other items visible on the stack, labelled "2:" and "3:".

The operation of the stack is conceptually described in the RPN Quick Start Guide. Also, specific operations for manipulating the stack on this calculator are described in the Working With The Stack document.

However, there are some things you can do from a UI perspective which helps to visualize the stack. First of all, you can swipe up and down to scroll up and down through the stack. If you have many items on the stack, you normally only see the top 3 or 4 items. But, with a quick swipe, you can inspect other items easily.

Also, you can use the "▪View↑" and "▪View↓" buttons to change what item appears at the bottom of the display. Normally, the top of the stack, labelled "1:" is shown at the bottom of the display. If you press the "▪View↑" button once, the bottom of the display will now show the item labelled "2:". The item at the top of the stack still exists, it is just not displayed. Continually pressing the "▪View↑" button will hide more and more items from the top of the stack and make items deeper in the stack visible. Pressing the "▪View↓" button reverses the affect. However, this isn't as convenient as just scrolling through the stack with your finger.

As described in the command line section, you can double tap on an item on the stack and then select "Copy" to copy that stack item to the clipboard. You can use that data within the calculator or in another application. If you double tap on a stack item and select "Paste", the contents of the clipboard are not pasted onto that stack item but are instead appended to the command line. All data entry is sent to the command line and not directly to items on the stack.

Finally, items on the stack may be truncated if they cannot fit on a single line. The exception to this is the item at the top of the stack (the bottom of the display). By default, that item will not be truncated and multiple lines of output will be used to display that item in full. That behaviour can be changed by using the ML operation (ML stands for multi-line) in the "▪Mode" menu. Also, you can use "▪View↑" and "▪View↓" to change the item shown at the bottom of the display, allowing you to see an non-truncated version of other items in the stack.

# Halcyon Calc - Working With Real Numbers

Halcyon Calc, like most calculators, is great for manipulating real numbers. This document will describe how to work with real numbers in the following sections:



For the purposes of this document, the assumption is that the radix character is a period (see the RDX, operation for more information), thus making the comma character the separator. If you have the calculator configured differently, swap the use of the period and comma wherever they occur.

## Entering Real Numbers:

There are a few different formats which the calculator understands which you can use in order to enter a real number:

1. If the number is zero, you can just hit the "0" button followed by "Enter"
2. If the number is a simple positive integer, you can just hit the buttons for the digits in the number. Start with the left-most digit and enter the numbers as they appear. For example, to enter the number 1230, hit the buttons "1", "2", "3", "0" and finally "Enter".
3. If the number is positive but has a fractional part, you can hit the buttons for the non-fractional part of the number, then the period followed by the fractional digits. Again, enter the digits left to right. For example, to enter the number 12.3, hit the buttons "1", "2", ".", "3" and finally "Enter"
4. If the number is positive but less than one, like 0.123 for example, you can enter that as ".", "1", "2", "3" and finally "Enter". You could also hit the "0" button first but it isn't required.
5. If the number is a negative integer, you must use the "CHS" button which stands for "CHange Sign". Note that if you do not have anything on the command line yet, pressing "CHS" will affect the item at the top of the stack. Specifically, it executes the NEG operation on the item from the top of the stack.

   But if you are entering a number, it will reverse the sign of that number. So, to enter the number -1230, hit the buttons "1", "2", "3", "0", "CHS" and finally "Enter". In fact, you can press "CHS" any time after you hit the "1" button. It will change the sign of the number and you can continue adding digits on the command line after that. Or, if you forget to the press "CHS" and complete entering the number as a positive integer, you can press the "CHS" button after to change its sign after pushing it onto the stack.

   If you press "CHS" a second time on a number on the command line, it will reverse the negative sign so the number is positive again. So, it is easy to fix if you make a mistake.

6. If the number is a negative number with a fractional part, you can enter the number as a positive number with a fractional part and use the "CHS" button to make it negative any time after starting entering the number. Similarly for a negative number with only a fractional part.

7. If the number is very large, it may be easiest to represent it in exponential or scientific notation. For example, if you want to enter the number 1.23 x $10^{30}$, you would hit the buttons "1", ".", "2", "3", "EEX", "3", "0" and finally "Enter". The "EEX" button is used to enter a number is exponential notation. When you press it, an "E" character will be be added to the command line which is short hand for "times ten to the power of...". So, pressing those buttons will result in a command line which looks like "1.23E30".

   Note that you can actually use the "E" button also from the left page of buttons instead of the "EEX" button. The command line does not distinguish between an "E" added by the "E" button or the "EEX" button. However, the "EEX" button is more convenient because it is on the same page as the digits themselves.

   Also, the number does not need to have a decimal point in it. You could enter "123E28" which actually ends up being the same number (try it if you like).

8. If the number is a very large negative number, again you can use scientific notation. In this case, you should press the "CHS" button after entering the first digit but before pressing "EEX". If you forget to press "CHS" before pressing "EEX", you can always press it after pushing the number on the stack.
9. If the number is very small, you can also use scientific notation. In this case, the exponent will be a negative number. If you want to enter the number 1.23 x $10^{-30}$, you would hit the buttons "1", ".", "2", "3", "EEX", "3", "0", "CHS" and finally "Enter". Note that the "CHS" button operates on the exponent and not the number itself in this case. You can press "CHS" any time after hitting the "EEX" button. If you forget to press the "CHS" button before entering the number, you will need to "▪Edit" the number again and apply the "CHS" to the exponent.
10. If the number is a very small negative number, like -1.23 x 10-30, you need to press "CHS" twice. You could enter this number by pressing "1", ".", "2", "3", "CHS", "EEX", "3", "0", "CHS" and finally "Enter". The first "CHS" makes the number itself negative while the second one makes the exponent negative. As before, you can press the first "CHS" any time after hitting the first digit and before pressing "EEX" while the second one must happen after pressing "EEX".

## Formatting Real Numbers:

Just like there are multiple ways to enter real numbers, you can display them on the calculator in multiple formats. These formats are global modes which means that all numbers displayed on the calculator will appear in a single format. If you change to a different format, all numbers displayed on the stack will automatically change and reflect the new format.

With any one format, not all digits will be displayed. In some formats, you can control exactly how many digits are shown. However, even though digits might not be shown they are still valid and used in calculations. You can round numbers to only the digits shown by using the RND operation if that is what you would like to do.

All of the buttons for setting the number format mode can be found in the Mode Menu which you can see by pressing the "▪Mode" button. On that menu, you are able to chose between the following formats:

- Standard - Select this by hitting the STD button from the Mode menu. In this mode,

numbers are displayed with up to 12 digits of accuracy. Exponential mode will not be used unless necessary. See the documentation for STD for more details.

- Fixed - In fixed mode, you can control the number of digits shown to the right of the decimal point. First, enter the number of digits you want used in fixed mode and then press the FIX button from the Mode menu. If you specify 3, then the number 50 will be displayed as "50.000". Zeroes will be added to the right of the decimal place as required. Just like in standard mode, exponential notation will be used when necessary. In exponential mode, the number of digits specified controls the display also. So, if the calculator is still in fixed mode with 3 as its number of digits, the number $5 \times 10^{20}$ will be shown as "5.000E20".
- Scientific - Just like fixed mode, scientific mode expects a number which represents the number of digits to show to the right of the decimal point. In this case, exponential format will always be used, unlike in standard and fixed mode. Enter the number of digits you want displayed to the right of the decimal place and press the SCI button from the Mode menu. If the number 5 is show in scientific notation with 3 decimal place digits, the calculator will display "5.000E0".
- Engineering - Engineering mode also always uses exponential notation but in this case, the exponent is always evenly divisible by 3. Like fixed and scientific modes, engineering mode takes a number which represents how many digits to show. In this case, the number of digits shown is the number specified plus one. Those digits might be to the left or to the right of the decimal place. Enter the number of digits you want displayed minus one and press the ENG button from the Mode menu. For example, if the number 12300 is shown in engineering notation with 3 set as the number of digits, the number will appear as 12.30E3. The exponent is evenly divisible by three and four digits are shown in total.

## Special Real Numbers:

There are two special real numbers which are built into the calculator. They are π (pi) and e. The calculator knows their value and gives you a shortcut which you can use to enter them.

In the case of pi, you can press the "▪π" button on the right page of buttons, followed by "Enter". You will see 'π' at the top of the stack. You can then operate on that value like you would any other real number, but instead of giving you a result, the calculator will build an "expression" (see the Working With Expressions guide for more information). That means if you add 1 to 'π' on the stack, you will see 'π+1'. At any time, you can press the "▪→Num" button to convert that expression into a number.

Similarly for the number e. To enter it, press the "e" button (make sure you are in lower case mode and if not, press the "LC" button first). You will see 'e' at the top of the stack. Again, you can use the "▪→Num" button to covert that to a number at any time.

Those are the only two real number constants built into the calculator but you can define your own. Refer to the Working With Symbols guide for more information.

## Limitations And Accuracy:

Real numbers in the calculator are stored in double precision floating point format. For people unfamiliar with what that means, it implies limits to how accurate the calculator can be. The symbols MAXR and MINR can help to understand the limits of the calculator. These symbols are the largest positive number and the smallest positive number which can be represented in a real number on the calculator. You can use the "▪→Num" button to convert these symbols to numbers

if you are curious what they are.

If your result is larger than MAXR or smaller than MINR, the calculator will not give you the result you expect. But, there are also situations where the value of a calculation may not be what you expect. In double precision format, a number can only be represented with up to 16 digits of precision. If you subtract two numbers which you believe to be different but are only different in the digits beyond the first 16, the result may end up being 0 even though you might have expected a non-zero answer. An example of this would be adding 1E20 and 1E-20 and then subtracting 1E20. Even though the answer should be 1E-20, the calculator will give 0 as its answer. The problem is that 1E20 plus 1E-20 is still 1E20 to the calculator. It would need more than 40 digits of precision to accurately add those two numbers. Instead the addition has no affect on the very large number.

For most uses, you don't need to be aware of these kinds of limitations and most calculators have them. But, if you are seeing results you don't expect, this might be why.

## Manipulating Real Numbers:

There are many operations which you can use when working with real numbers. The standard addition, subtraction, multiplication and division operations are obvious. Here are some links to reference material which are relevant for real numbers:

- The Real Menu has several useful operations for working with real numbers.
- A list of operations which take real numbers can be found here.
- A list of operations which produce real numbers can be found here.

# Halcyon Calc - Working With The Stack

This document will build on the information from the section about the stack in the Working With The UI guide. However, that document was more about navigating the stack. This document is about manipulating the stack which is very important for proficient use of an RPN calculator.

Some of the best tools for working with the stack allow you to recover from mistakes. If you push something onto the stack which you didn't want or maybe you don't need that item on the stack any longer, use the "Drop" button to remove the item from the top of the stack. The "Drop" button executes the DROP operation so you can refer to the reference documentation for the operation for more information. Maybe the stack has many items on it and you want to clear the whole stack. You could press "Drop" multiple times but a faster way to accomplish that is the press the "▪Clear" button. That button will execute the CLEAR operation and the stack will be empty after executing it.

Maybe you have performed an operation by accident and you would like the previous values back. You can press the "▪Undo" button and the stack will be returned back to the state it was in prior to the previous operation. You cannot undo multiple operations, just the most recent operation. An alternative is to press the "▪Last" button which will push onto the stack the arguments passed in to the most recent operation. The "▪Last" button executes the LAST and allows you to preserve the result of the previous operation and get its arguments back onto the stack.

Often you want to copy the item at the top of the stack. That allows you to preserve that value and perform an operation on it. To copy the item from the top of the stack, you can use the DUP operation from the Stack menu. After executing that operation, a copy of the item on the top of the stack will be pushed, leaving two instances of that item at the top two positions on the stack. However, as a shortcut, you can press the "Enter" button with nothing on the command line which will execute the DUP operation.

Many times you will find the the item just below the top of the stack is the one you want to operate on next. You could "Drop" the item at the top of the stack but what if you want to preserve that item? In that case, you can press the "Swap" button. The "Swap" button executes the SWAP operation and will pop the top two items and then push them back on in reverse order.

What if the item you want to access is not next to the top? In that case you can use the "▪Roll" button. Push the number of the item you want to retrieve. Imagine you have the numbers 50, 30 and 20 on the stack and 50 is the top of the stack, 30 below that and 20 next. If you want 20 to be on the top of the stack, push "3" onto the stack and press the "▪Roll" button. That will execute the ROLL operation and leave the stack with 20 at the top, following by 50 and finally 30.

There are many more operations which you can use to manipulate the stack but these are the most frequently used. For more information about stack operations, check the Stack menu.

# Halcyon Calc - Working With Complex Numbers

Halcyon Calc can manipulate complex numbers and most operations which you can use with real numbers also works with complex numbers. For people unfamiliar with complex numbers, a complex number consists of two components. There is a real component and an imaginary component. So, a complex number looks like two real numbers jammed together.

This document will describe how to work with complex numbers on Halcyon Calc in the following sections:



For the purposes of this document, the assumption is that the radix character is a period (see the RDX, operation for more information), thus making the comma character the separator. If you have the calculator configured differently, swap the use of the period and comma wherever they occur.

Also, complex numbers are always assumed to be in rectangular coordinates and not polar coordinates. There are operations which convert from rectangular to polar coordinates and back. However, if you perform any other operation on a complex expressed in polar coordinates, you will not get the answer you expect. Instead, convert back to rectangular, perform your operation and return to polar coordinates.

## Entering Complex Numbers:

Because a complex number has two components which each look like a real number, it is worth reviewing how to enter real numbers. A complex number looks like:

```
(X,Y)
```

where X and Y entered just like real numbers. So, to enter the complex number (1.2,3.4), you would hit the buttons "(", "1", ".", "2", ",", "3", ".", "4", "▪)" and finally "Enter". Actually, you can drop the "▪)" button. You don't need to close the parentheses. The calculator will still interpret the content as a complex number.

And you can use the "CHS" and "EEX" buttons to change the sign or use exponential notation for one or both components of the complex number.

## Formatting Complex Numbers:

The two numeric components in a complex number are formatted using the same rules used with formatting real numbers. When you switch formatting modes, real numbers and complex numbers will be affected by that formatting mode. Refer to the documentation for real numbers for more details.

## Special Complex Numbers:

There is one special complex number built into the calculator. The special number, i which represents an imaginary unit (where 1 represents a real unit) in the complex plane is built in. The calculator knows its value and gives you a shortcut which you can use to enter it.

To enter "i", you can press the "i" button on the left page of buttons (make sure you are in lower case mode and if not, press the "LC" button first), followed by "Enter". You will see 'i' at the top of the stack. You can then operate on that value like you would any other number, but instead of giving you a result, the calculator will build an "expression" (see the Working With Expressions guide for more information). That means if you add 1 to 'i' on the stack, you will see 'i+1'. At any time, you can press the "▪→Num" button to convert that expression into a number.

This means you can also enter a complex number as an expression in terms of "i". To do so with the complex number (5,4), you could enter:

```
'5+4*i
```

This is equivalent to (5,4), however this item on the stack is an expression, not a complex number. But, if you hit the "▪→Num" button, the expression will be converted into the equivalent (5,4) complex number.

This is the only complex number constant built into the calculator but you can define your own. Refer to the Working With Symbols guide for more information.

## Manipulating Complex Numbers:

There are many operations which you can use when working with complex numbers. The standard addition, subtraction, multiplication and division operations work on complex numbers but so do many other operations. Here are some links to reference material which are relevant for complex numbers:

- The Complex Menu has several useful operations for working with complex numbers.
- A list of operations which take complex numbers can be found here.
- A list of operations which produce complex numbers can be found here.

# Halcyon Calc - Working With Arrays

Halcyon Calc can manipulate matrices and vectors. Collectively, matrices and vectors are referred to as "arrays" in Halcyon Calc. A vector is "N" real or complex numbers making up a one dimensional array of values. A matrix is "N" x "M" real or complex numbers making a two dimensional array of values.



## Entering Vectors and Matrices:

Because vectors and matrices are a collection of one or more real or complex numbers, it is worth reviewing how to enter real numbers and how to enter complex numbers. A real vector can be entered like this:

```
[A B C]
```

where A, B and C are entered just like real numbers. This creates a vector with three values. A vector must have at least one value and can have many more than three. Note that you can use a comma (or a period of the radix character is set to comma) to separate values instead of space. Also, the closing square bracket is not required if this is the last value being entered.

Similarly, a complex vector can be entered like this:

```
[(A,B) (C,D) (E,F)]
```

where A, B, C, D, E and F are entered just like real numbers. This creates a complex vector of three values but you can create vectors with more or fewer items. As above, comma can be used as a separator and the final square bracket is optional if this is the end of the line. In fact, the final round bracket can be left off if "F" is at the end of the line. You can mix pure real numbers and complex numbers when entering a vector. When parsed by the calculator, if there is at least one complex value in the vector, the entire vector will appear as complex values.

A real matrix can be entered like this:

```
[[A B C][D E F]]
```

where A, B, C, D, E and F are entered just like real numbers. This creates a real matrix with two rows and three columns. You can use commas instead of spaces to separate between values. Also, the closing square brackets are optional. So, this can be entered as:

```
[[A B C[D E F
```

A complex matrix can be entered like this:

```
[[(A,B) (C,D) (E,F)][(G,H) (I,J) (K,L)]]
```

where A through L are entered as real numbers. This creates a complex matrix with two rows and three columns. As before, commas can be used instead of spaces between items. The closing square brackets and even the final closing bracket is optional:

```
[[(A,B) (C,D) (E,F)[(G,H) (I,J) (K,L
```

## Formatting Vectors and Matrices:

Because vectors and matrices are made up of several real or complex values, they can be very long and difficult to read. However, if you don't all digits of precision in the output, you may want to change number formatting to make the display more readable. Formatting real numbers describes how to do this. When you switch formatting modes, real numbers, complex numbers, vectors and matrices on the stack will automatically be displayed in the new formatting mode.

## Manipulating Vectors and Matrices:

There are many operations which you can use when working with vectors and matrices. The standard addition, subtraction, multiplication and division operations work but so do many other operations. Here are some links to reference material which are relevant for complex numbers:

- The Array Menu has several useful operations for working with vectors and matrices.
- A list of operations which take vectors or matrices can be found here.
- A list of operations which produce vectors or matrices can be found here.

# Halcyon Calc - Working With Integer Numbers

Halcyon Calc can manipulate integer numbers. This document has the following sections to help you work with integers on the calculator:



## Entering Integer Numbers:

An integer number can be entered in one of four different forms:

- Decimal - In decimal mode, you use the usual digits 0 through 9. The number is expressed in base 10.
- Hexadecimal - In hexadecimal mode, you use the digits 0 through 9 and the letters A through F (which represent the numbers 10 through 15). The number is expressed in base 16.
- Octal - In octal mode, you use the digits 0 through 7. The number is expressed in base 8.
- Binary - In binary mode, you use the digits 0 and 1. The number is expressed in base 2.

In all cases, you start entering an integer by pressing the "#" button. All integers start with this symbol and indicates that the digits which follow should be interpreted as an integer. After the "#" symbol, you then hit the digits you want to enter, left to right in your number. The digits you enter depend on the base of the number you are entering. Note that if you are entering a hexadecimal number, the letters A through F must be entered in upper case. The calculator will not understand the number if you use lower case.

Finally, you add a single character to indicate the base of the number you are entering. Hit the "d" key to indicate the number is a decimal number. Use the "h" key to mark it as hexadecimal or the "o" key if it is octal. Finally, you use the "b" key to indicate the number should be interpreted as binary. Note that the character which controls the base of the number must be entered in lower case.

The calculator has an integer mode which controls which base is used for formatting integer numbers (see the formatting section for more information). But, you can omit the final character which specifies the base of the number you are entering if you are in that mode. So, if you enter a hexadecimal number, you do not have to put an "h" at the end of the number if you happen to be in HEX mode.

To demonstrate how to enter integers, imagine you want to enter the decimal number 452 as an integer. The following table describes how to do that in all of the four different bases supported:

- **Decimal** - #452d
- **Hexadecimal** - #1C4h
- **Octal** - #704o
- **Binary** - #111000100b

Note that the number you see on the stack may not match exactly what you entered in some

cases, because the word size may result in some digits not being shown. The word size and how it affects integers is explained in the formatting section.

## Converting To Or From Integer Numbers:

Often, you may have a real number on the stack which you would like to manipulate as an integer number. In that case, you can use the R→B operation from the Binary menu. This operation will take a real number from the top of the stack and pushes the equivalent integer onto the stack. Another way to accomplish this is to add "#0" to the real number on the stack. The + operation can take a real number and an integer number and produces an integer result. By adding an integer zero, you are effectively converting the real number to an integer.

The alternate conversion is possible also. If you have an integer at the top of the stack which you would like to convert to a real number, you can use the B→R operation from the Binary menu.

## Formatting Integer Numbers:

There are two key parameters which drive how an integer is formatted. Firstly, the base mode the calculator is in determines what base is used to display integers on the stack. The DEC, HEX, OCT and BIN operation from the Binary menu will set the calculator to the decimal, hexadecimal, octal or binary mode respectively.

The calculator also has a word size which determines how many binary digits are valid. By default, the calculator runs with a word size of 64 which means you can enter integer values between 0 and $2^{64}$-1. The word size can be set to any number between 1 and 64 and you would use the STWS (SeT Word Size) operation from the Binary menu to set the work size. Push the new word size onto the stack as a real number and execute STWS. If you don't know what the word size is currently, execute the RCWS (ReCall Word Size) operation from the Binary menu. That operation will push the current word size onto the stack as a real number.

Regardless of the base you are in, the word size affects how many digits you will see. If you decrease the word size, integers on the stack may lose their upper digits and one digit may change value (upper bits in that digit were masked out). If you increase the word size, integers on the stack may regain their upper digits.

Because integers are the primary data manipulated by microprocessors/computers, a common use for integers on the calculator is to perform these kinds of calculations. If the computer you are working with has a 32 bit word size, then it is probably natural to set the word size on the calculator to match.

## Manipulating Integer Numbers:

There are many operations which you can use when working with integer numbers. The standard addition, subtraction, multiplication and division operations are obvious. Here are some links to reference material which are relevant for integer numbers:

- The Binary Menu has several useful operations for working with integer numbers. Here, you will find shift and rotate operations and the standard logic operations like and, or, exclusive or etc.
- A list of operations which take integer numbers can be found here.

- A list of operations which produce integer numbers can be found here.

# Halcyon Calc - Working With Lists

Halcyon Calc can group items together into lists which can help you to keep related items in one place. This document has the following sections to help you work with lists on the calculator:



Note that you might think you could use lists as a vector or even a matrix. However, a list really is not intended to be used as these mathematical concepts. Instead, lists are intended to gather together different items, each might be different types of items. So, your list can have real numbers, complex numbers, symbols, expressions or even other lists in them, all at the same time.

## Entering Lists:

A list is a series of items surrounded by opening and closing curly braces. So, to enter a list of the numbers 1, 2 and 3, you can hit the buttons "{", "1", ",", "2", ",", "3", "}" and finally "Enter". The closing brace is not required and if it isn't entered, the calculator will still parse the command line as a list. In this example, comma was used as a separator but that depends on the radix mode on the calculator. If the radix is comma, then you can use period to separate items in the list. Also, you can separate items with a space or a new line character.

The items in the list should be entered according to the rules of the items being added. So, you can use all of the rules for entering real numbers, complex numbers, integers, etc for the items in your list. Also, a list can be an item within another list. Any item you can put onto the stack on the calculator can be an element of a list.

## Converting To Or From Lists:

Often, you may have a series of items on the stack which you would like to gather together as a list. In that case, you can use the →LIST operation from the List menu. This operation will take a real number from the top of the stack which is the number of other items to grab from the stack and put into a list. It then pushes the list containing those items onto the stack.

The alternate conversion is possible also. If you have a list at the top of the stack which you would like to break out into its individual items, you can use the LIST→ operation from the List menu. This operation pops the list from the stack and then pushes each item from the list in order. Finally, it pushes the number of items which were in the list onto the stack as a real number.

## Formatting Lists:

A list has no specific formatting options, however any real, complex or integer numbers in the list will be affected by the formatting modes which affect those types of items.

## Manipulating Lists:

There are many operations which you can use when working with lists. You can use + to add an item to a the beginning or end of a list or to concatenate two lists together. Here are some links to reference material which are relevant for lists:

- The List Menu has several useful operations for working with lists.
- A list of operations which take lists can be found here.
- A list of operations which produce lists can be found here.

# Halcyon Calc - Working With Strings

Halcyon Calc operate on strings of characters, or "strings" for short. The following sections describe how to work with strings on the calculator:



## Entering Strings:

A string is entered as a series of characters surrounded by double quote characters ("). To start entering a string, hit the "▪"" button. Then, you can type whatever characters you want to be a part of the string. Optionally, you can finish the string with a hit of the "▪"" button and "Enter", or just hit "Enter". The final double quote is assumed if you do not enter it.

When you hit the first ▪"" button, the calculator goes into "alpha" entry mode. For more details of what that means, refer to the Command Line section of the UI Guide.

## Converting To Or From Strings:

You can take any item from the top of the stack and convert it to a string by using the →STR operation from the String menu. This operation will take the item from the top of the stack and then push a string to onto the stack whose contents is the character representation of that number (ie how it appeared on the stack prior to this operation but with double quotes around it).

The alternate conversion is possible also. If you have a string at the top of the stack, you can use the STR→ operation from the String menu to convert it to the item or items it contains. This operation pops the string from the top of the stack. Then, the calculator parses that string as though it was just typed at the command line. The result of parsing the contents of the string is then pushed onto the stack. So, the string "123" will result in the real number 123 being pushed onto the stack.

## Formatting Strings:

There are no formatting options in the calculator for strings. Also, any numbers which appear in the string are not affected by the different formatting modes for numbers. The contents of a string is unchanged by these modes.

## Manipulating Strings:

There are many operations which you can use when working with strings. You can use + to concatenate two strings together. Here are some links to reference material which are relevant for lists:

- The String Menu has several useful operations for working with strings.
- A list of operations which take strings can be found here.
- A list of operations which produce strings can be found here.

# Halcyon Calc - Working With Symbols

Halcyon Calc can store values into symbols and manipulate those symbols. Working with these symbols is described in the following sections:



## Entering Symbols:

A symbol is a name which can be used to store a value for future use. The symbol name is between 1 and 127 characters long and must not start with a digit. Valid characters in the symbol name are any letter (upper or lower case), digit (as long as it isn't the first character) or one of the following:

- ?
- →
- ∑
- °
- µ
- π

To enter a symbol on the command line, press the "'" (single quote) button, then hit each button for each character in the name of the symbol, and finally press "Enter". You can also optionally press the "'" button again at the end of the symbol name to close the quotation. If you are entering the symbol "X", you can do that by pressing "'", "X" and finally "Enter". You will find that 'X' will appear on the stack which is a symbol.

Note that when you press the "'" button the first time, the calculator goes into "algebraic mode". When in algebraic mode, pressing buttons which normally execute operations will instead insert text onto the command line. For example, pressing the "SIN" button will append "SIN(" onto the command line. Refer to command line section of the UI Guide for more information about algebraic mode.

Alternatively, if you skip pressing the "'" button and just press "X" and "Enter", the calculator does the following:

- The calculator searches its symbol table to see if the symbol has a value. If so, it pushes that value onto the stack and not the symbol itself. So, in this example, the calculator searches its symbol table for X and if it finds a value, it does not push 'X' but instead pushes the value of X.
- If the symbol does not have a value, then the symbol itself is pushed onto the stack. So, in this example, 'X' will be pushed onto the stack if X has no value currently.

## Storing Values In Symbols:

To store a value in a symbol, push the value you want to store on the stack followed by the symbol you would like to store it into. Then, press the "Sto" button which will pop those two items from the stack and store that value into that symbol. The "Sto" button will execute the STO operation.

Note that anything which can be pushed onto the stack can be stored in a symbol. Not only real and complex numbers, but lists, strings, expressions and even other symbols can be stored in a symbol.

A common use for symbols might be to declare your own constants. Perhaps you often need to do calculations with Avogadro's constant. If so, you can store the value 6.02214179E23 into the symbol "N". Then, whenever you need to work with that constant, you can just recall the value of "N".

## Recalling The Values Of Symbols:

There are two ways to recall the value of a symbol. The easy way is to enter the name of the symbol at the command line without prefixing it with a single quote. An unquoted symbol name is looked up and if a value is found, the value is pushed onto the stack. Otherwise, the symbol itself is pushed onto the stack.

The alternative is to push the symbol onto the stack and then press the "▪Rcl" button. Pressing this button will execute the RCL operation and pop the symbol from the stack and replace it with the value of that symbol. If the symbol has no value associated with it in the symbol table, an error will be displayed.

## Clearing The Values Of Symbols:

If you no longer need the value of a symbol and would like to clear the symbol from the symbol table, use the "▪Purge" button. To purge the value of a symbol, push the symbol name onto the stack and press the "▪Purge" button. This button executes the PURGE operation and it pops the symbol name from the stack and removes that symbol from the symbol table. Any value associated with that symbol is lost.

## Browsing Symbols:

If you don't recall the symbols in the symbol table or would like to quickly recall the value of several symbols, press the "User" button. This will cause a special set of menu buttons to slide in. Each page of six menu buttons will show a symbol name (truncated if necessary to fit on the button). You can swipe to the left and right to view the different pages of buttons or use the "Next" and "▪Prev" buttons to flip through the pages. This is a great way to review which symbols you have created.

When you press one of these buttons, the value of that symbol will be recalled and pushed onto the stack. Also, if you press "'" first (or put the calculator in algebraic mode), then the symbol name will be appended to the command line. If the button you press is labelled "X", then the command line will now read 'X. So, pressing "Enter" will push the symbol name onto the stack (in

this case, 'X' will be pushed onto the stack). Using these two methods, the "User" menu is a great way to get the value of a symbol or to push the symbol name itself onto the stack.

## Organizing Symbols:

By default, the newest symbols created are shown earlier in the "User" menu buttons. The oldest symbol created will be on the last page. But, you may find this default order difficult to use. If so, you can use the ORDER operation from the "Memory" menu to tweak the order of symbols. First, create a list of symbol names in the order you would like them to appear. You don't need to include all symbol names and any left out of the list will be put at the end. Push the list of symbols onto the stack and then execute the ORDER operation. After execution, the list will be popped from the stack and the symbols in the "User" menu will be re-ordered.

If you want a list to start with, you can execute the VARS operation from the "Memory" menu. This operation will push a list of symbols onto the stack in the order they appear right now. You could then edit that list and re-order some symbols to prepare it for use with ORDER operation.

But, in some situations, you really would like to keep a group of symbols together and separate them from other symbols. For this, you can create directories of symbols. Note that directories are not supported on Halcyon Calc Lite. By default, the calculator starts with a single root directory called "HOME". If you want to create a directory called DIR1, you push the symbol 'DIR1' into the stack and then execute the CRDIR operation (CReate DIRectory) from the "Memory" menu. The operation will pop that symbol name from the stack and create a directory with that name in the current directory. It does not change the current directory.

To change to a directory, you can type the symbol for that directory into the command line without the "'" prefixing it. When the calculator looks up the "value" of the symbol, it finds that the symbol is a directory and then switches to that directory. Alternatively, if you bring up the "User" menu and press the symbol button for that directory, then the calculator switches to that directory.

When the calculator switches to a directory, the following things happen:

- When you press the "User" menu button, you only see the symbols which exist in that directory.
- When you store a value in a symbol, that symbol will be created in the current directory.
- When you look up the value of a symbol, the calculator checks the current directory first. If it finds the symbol there, it recalls that value. If not, it checks the "parent" directory of the current directory. It continues checking until it gets to the "HOME" directory and if it is still not found, then the symbol lookup fails.

The lookup behaviour means that some good rules of thumb when using directories are:

- If you want to have a symbol with two different values, you can store one value in one directory and put the other value in the other directory. Do not create one of these directories in the other. Instead, create them both in the "HOME" directory for example. Then, switch into one of these directories and perform calculations with one value. When you want the other value, switch to the other directory.
- If you have a single value which you want to be accessible in two different directories, put them in a common parent. A good place might be in "HOME". For example, physical constants might be a good thing to store in "HOME" since they always will have that value.

At any time, you can use the PATH operation from the "Memory" menu to get the current directory path. This operation will push a list onto the stack which starts with the symbol HOME and each successive symbol is the next directory which eventually leads to the current directory the calculator is set to. Also, if you want to descend to a different directory, you can execute the HOME operation from the "Memory" menu to return back to the "HOME" directory. From there, you can descend through to a different directory.

## Custom Menus:

With the menu buttons which dynamically change, the calculator can present many more operations than can actually fit comfortably on the screen. But, you may find that you end up switching between many different sets of menu buttons or flipping through pages often to get to the four or five buttons you mainly access. Also, you may have a large number of symbols stored in the symbol table, making the "User" menu tough to navigate but it would be great to have a smaller selection of symbols in a more convenient menu.

The calculator offers the MENU operation from the "Memory" menu to allow you to create a custom set of menu buttons. First, create a list of symbols which you want to have accessible. To create a list of symbols, you want to leave off the "'" character. So, to make a menu of three trig operations, enter the list:

```
{ SIN COS TAN
```

into the command line. You can enter the names of operations or symbols. Once you have your list, execute the MENU operation which pops that list off the stack and creates the custom menu for you. To access that custom menu, press the "▪Custom" button.

## Manipulating Symbols:

There are many operations which you can use when working with symbols:

- The Store Menu has several useful operations for working with symbols. These operations can be used to perform operations on the values of symbols by symbol name alone.
- The Memory Menu has several useful operations for organizing symbols.
- A list of operations which take symbols can be found here.
- A list of operations which produce symbols can be found here.

# Halcyon Calc - Working With Expressions

One of the key features of Halcyon Calc is its ability to manipulate symbolic expressions. Working with these expressions is described in the following sections:



## Entering Expressions:

There are two fundamental ways to enter an expression. You can either type the expression in directly into the command line or you can put your symbols and numbers onto the stack and execute the operations you want on them, slowly building up an expression. In fact, you can mix these two methods together when building very large, complicated expressions.

To enter an expression at the command line, start your expression with the "'" character and then hit the appropriate buttons to type the expression you want. When you start the expression with a single quote, the calculator will switch to algebraic mode (as described in the command line section of the UI Guide). You can rely on that behaviour to assist you when building expressions.

By way of an example, imagine that you want to enter the expression: $X^3 * \sin(Y)$. To enter this on the command line, press "'", "X", "■^", "3", "x" (multiply), "SIN", "Y" and finally "Enter". The assumption here is that the "Trig" menu is already displayed and the "SIN" button is visible. When you press the different button operations in this example ("■^", "x" and "SIN"), the calculator did not execute the operation but instead appended the operation onto the command line. Before pressing "Enter", the command line should look like:

```
'X^3*SIN(Y
```

Note that the closing bracket and the closing single quote is left out. The calculator will fill those in at the end of the expression as required but you can provide them also. After pressing "Enter", the stack will show the expression:

```
'X^3*SIN(Y)'
```

The alternate way to build an expression is to push the symbols and numbers onto the stack in the order you want them evaluated and apply the operations to them. So, to build the same expression this way, you could do the following:

1. Push the symbol 'X' onto the stack.
2. Push the number "3" onto the stack.
3. Execute the "■^" operation. This will pop those two items from the stack and perform the power operation on them. Because 'X' is a symbol, the result is an expression in X.

Specifically, 'X^3' is pushed onto the stack.
4. Push the symbol 'Y' onto the stack.
5. Execute the "SIN" operation. This will pop 'Y' off the stack and perform the sine operation on that symbol. But, because it is working with a symbol, the result is an expression in Y. Specifically, 'SIN(Y)' is pushed onto the stack.
6. Finally, execute the "x" (multiply) operation. This will pop 'X^3' and 'SIN(Y)' from the stack and perform the multiply on these items. Because both are expressions, the result will also be an expression. Specifically, 'X^3*SIN(Y)' is pushed onto the stack.

The interesting thing is that entering an expression at the command line uses the normal "left to right" syntax which people are used to for mathematical expressions. You can take advantage of this and enter expressions without any symbols if you prefer to work in a non-RPN mode. So, if you want to multiply 10.32 by 23.74, you could enter the expression '10.32*23.74' and then evaluate it. Building an expression one operation at a time is essentially the RPN way of working where you put the arguments you want onto the stack and then operate on them. In this case, some or all of those arguments are symbols or other expressions. Which you use depends on what you find most convenient.

## Operator Precedence:

When looking at an expression, it may not be immediately obvious which operations occur in which order. The precedence rules determine this order. The higher the number in the following list, the higher the precedence. An operation with higher precedence will be evaluated first. Operations with equal precedence are evaluated from left to right. You can use brackets to override this evaluation order since the sub-expression in the brackets is evaluated first. The operator precedence order is:

1. ≡
2. OR, XOR
3. AND, NOT
4. ≤, ≥, ≡≡, ≠, ≦, ≧
5. + (add), - (subtract)
6. x (multiply), / (divide), - (unary negation)
7. ^ (power), √
8. All other operations not listed here.

An example of unary negation might be '-X' which means the value of X negated. The symbol is the same as subtract but the calculator can tell whether the operation is unary negation or subtraction from the context.

When entering an expression into the command line, make sure you use brackets as necessary to ensure that the expression is evaluated in the order you intend. If you want an operation with lower precedence evaluated before an operation of higher precedence, you must put the operation with lower precedence and its arguments in brackets. If you are building the expression by applying operations to symbols and numbers on the stack, the calculator adds brackets as necessary to ensure that the expression is evaluated in the order in which you execute the operations while building the expression.

## Equations:

An expression with a ≡ operation in it is special and is an equation. The equation describes a relationship between the left side and the right side of the expression. When you apply operations to an equation, the calculator does things differently.

Imagine you have the equation 'X=Y' on the stack and then push the number 3. Then, you execute the + operation. The operation is adding 3 to the equation 'X=Y'. What the calculator does is add three to both sides of the equation to preserve the relationship and pushes the result 'X+3=Y+3'.

Instead, if you have two equations on the stack, 'A=B' and 'C=D' and then execute the + operation, the calculator produces the result 'A+C=B+D'. It adds the left side of each equation and the right side of each equation to produce a result which preserves the relationship. In these examples, the + operation is used to illustrate the point. This works for any operation you can apply to an expression.

## Evaluating Expressions:

Once you have created an expression, you probably would like to calculate the value of that expression for some specific values of the symbols. There are three different ways to do so. The first method is to use the "Eval" button. To do this, you first need to store the values you want into the symbols involved. The values can be real numbers, complex numbers or just about anything else. In fact, you can store another expression into one or more of these symbols which would allow you to substitute that new expression for that symbol in the target expression.

Once you have stored the values you want, make sure the expression you want to evaluate is at the top of the stack and press the "Eval" button. That will execute the EVAL operation which pops the expression off of the stack and then looks up the value of any symbol which appears in the expression. If a symbol has a value, its value is substituted into the expression. If the value is itself another symbol or expression, that value is not evaluated. It is just substituted. If the symbol it looks up has no value, then the symbol will remain in the expression. If one of the global constants on the calculator appears in the expression (e, i and π), they will **not** be replaced with their value but will remain in the expression in symbolic form.

Then, any operations which now have non-symbolic inputs will be evaluated. If possible, the result of "Eval" will be a numeric value. But, if some symbols remain, the result will still be an expression.

Alternatively, you can press the "▪→Num" button which executes the →NUM operation. It does almost the exact same thing as executing "Eval", except any global constants (e, i and π) will be replaced by their numeric value. So, use "Eval" if you would like these global constants to remain in the expression and use "→Num" if you want a numeric result.

But, that can make evaluating an expression for different input values difficult. You must store your different input values in each symbol and then evaluate the expression. Also, the expression is popped off the stack so if you did not leave a copy on the stack, you will have to recall it from a variable (if you stored it in one), recall it using "Last" or re-enter it. A simpler way is to use the solver.

The first step in using the solver is to build the expression you would like to work with and put it on the top of the stack. Then, press the "Solv" button, opening the solve menu. Execute the "STEQ operation. This takes the expression from the top of the stack and stores it into a variable

called 'EQ', short for equation. Then, execute the "SOLVR operation to use the solver on that equation.

In the solver, you will find that the menu buttons have changed. First, you will see a button for each symbol which appears in the expression. If the expression is an equation, you will then see a "LEFT=" and "RIGHT=" button. If the expression has no = operation in it, then you will just see an "EXPR=" button.

To set a symbol to a particular value, push that value onto the stack and then press the button for the symbol. On the button press, the value is popped off of the stack and stored into that symbol. Do that for each symbol you would like to set. If you are working on an equation, you can press the "LEFT=" and "RIGHT=" to evaluate the left and right hand side of the equation. If the expression is not an equation, you can press the "EXPR=" button to evaluate that expression for those values. The result will be numeric if each symbol has a numeric value. If one or more symbols do not have a value, the result will be an expression in terms of those symbols.

In this mode, you can quickly change the value of your input symbols and generate results for different inputs quickly. Later, you will see how to find the roots of expressions in the solver also.

## Solving Quadratic Equations:

A quadratic polynomial can be solved by the calculator directly. To do so, push the expression which is quadratic in some symbol onto the stack. Then push the symbol you would like to solve for and execute the QUAD operation from the "Solv" menu. The roots of the quadratic will be pushed onto the stack as an expression. The result has a positive and a negative root which is expressed in the single expression by using a "s1" symbol. By setting "s1" to 1, the positive root can be evaluated and setting it to -1, the negative root can be found.

As an example, if you solve this quadratic equation in X:

```
'A*SQ(X)+B*X+C'
```

the calculator will push this result:

```
'(-B+s1*√(SQ(B)-4*A*C))/(2*A)'
```

That is the canonical solution for the roots to the quadratic equation, where s1 represents the ± which appears in the solution.

Note that it will also work with an expression which looks like '(X-3)*(X+8)'. The expression can contain non-polynomial operation like trig and logarithms as long as they are not evaluated in terms of the symbol being solved for. So, '(X-3)*(X+8)/SIN(Y)' is still a quadratic in X (but not in Y).

Finally, QUAD will also work on any polynomial of degree two or more. However, it treats any polynomial as a quadratic and discards the higher degree terms. The answer it provides in those cases is probably not too useful.

## Isolating A Symbol In An Expression:

In many cases, you would like to re-arrange an expression so that an expression of some

variable (for example, X) is expressed as X=something. The ISOL operation from the "Solv" menu allows you to do this. The key restriction is that the symbol must appear only once in the expression for the result to be an expression independent of that symbol. If the symbol being isolated appears multiple times, only its first appearance in the expression is isolated so the result will still have that symbol in it. For example, the equation 'X+X=3' would result in '3-X', isolating the first X which appears.

If the expression being operated on is not an equation, then an "=0" is appended to make it an equation.

To execute the operation, push the target expression and then the symbol being isolated on the stack. After executing ISOL, those two values will be popped off of the stack and the resulting expression will be pushed onto the stack.

The other restriction is that all operations which must be inverted to isolate the symbol but be "invertible". For example, the inverse of + is -. The inverse of ASIN is SIN. Some inverse operations are expressions and may include "s1", "s2", etc symbols to represent +1/-1 possibilities. Or, they may contain "n1", "n2", etc symbols which represent a periodic solution where you can evaluate them an an infinite number of integer values.

To find a list of invertible functions, refer to the Invertible Operations page.

## Finding Roots Of An Expression:

A common operation on an expression is to find a value for a symbol for which the expression evaluates to zero. These are call roots of the expression and the calculator can be used to find these roots. You can either use the ROOT operation from the "Solv" menu or use the solver described in the section on evaluating expressions.

To use the ROOT operation, push the expression you are working with onto the stack, then the symbol for which you would like to find a root, and finally a guess where that root may lie (a great way to find good guesses is to plot the expression - refer to the Working With Plots guide for more information). At a minimum, provide a single real number which is likely near the root. Better, provide a list of two or three real numbers around the root. The algorithm should perform better with more guesses. After executing the operation, the root will be pushed onto the stack. In some cases, the algorithm may run for a very long time and never converge onto a root. In that case, you can press the "Attn" button to interrupt execution. Or, the algorithm may result in an error saying it could not find a root. It could be there is no real root or maybe you should try to improve your guesses.

You can also use the solver to find roots. First, push your guess(es) onto the stack. If you have a single guess, push that real number. If you have two or three guesses, push a list of those real numbers onto the stack. Press the button representing the symbol you would like to solve for to store that guess into the symbol. You do want to store the list into the symbol if you are providing multiple guesses. The press the "▪" (red shift) button and then the button representing the symbol you would like to solve for. By pressing "▪<symbol>", you are instructing the calculator to find a root for that symbol. Again, the search for the root may not be successful and you may have to interrupt it with "Attn" or the calculator may produce an error. If so, refine your guesses and try again. Hopefully, the value of the root will be pushed onto the stack and the calculator will pop up a message to say a root was found. That value is also stored in the symbol itself so you can now evaluate the expression and see that it evaluates to zero (or very nearly zero).

## Finding Maxima And Minima Of An Expression:

The solver can be used to find a maximum or a minimum of an expression. To search for a max or min, you must provide a list of three guesses (a great way to find good guesses is to plot the expression - refer to the Working With Plots guide for more information). The lowest and the highest guess must be on either side of the maximum or the minimum and the middle guess should be closer to the max or min than the other two. Create a list with those guesses in it. The guesses can appear in the list in any order.

Press the button representing the symbol you would like to search for a maximum or a minimum in order to store that guess into the symbol. The press the "■" (red shift) button and then the button representing the symbol you would like to search. By pressing "■<symbol>", you are instructing the calculator to search for a maximum or a minimum. Note that this is the same method used to search for roots. As long as the three guesses are provided and those guesses surround a maximum or a minimum, the algorithm should return that maximum or minimum. If fewer than three guesses are provided or the guesses do not surround a max or min value, the calculator will search for a root.

The search for the maximum or minimum may not be successful and you may have to interrupt it with "Attn" or the calculator may produce an error. If so, refine your guesses and try again. Hopefully, the calculator does find a maximum or minimum value in which case it pushes the value for the symbol where the max or min was found and pops up a message to say what was found. Also, that value is stored into the symbol replacing the guess which was there before. Evaluate the expression to get the its value at the maximum or minimum found.

Note that this is a local maximum or minimum point. There may be other maxima or minima and there is no guarantee that this is the global maximum or global minimum for the expression.

## Manipulating Expressions:

There are many operations which you can use when working with expressions:

- The Solv Menu has several useful operations for working with expressions.
- A list of operations which take expressions can be found here.
- A list of operations which produce expressions can be found here.
- A list of operations which are valid in expressions can be found here.
- A list of operations which are invertible and thus can be used with the ISOL operation.

# Halcyon Calc - Working With Plots

Halcyon Calc can plot your expressions and equations in an X/Y axis allowing you to quickly see how that expression behaves over a series of values. Interact directly with your plots to focus in on the area you are interested in. Working with plots is described in the following sections:



## Specifying The Expression To Plot:

In order to plot an expression, you must first construct that expression on the stack. For information on entering an expression, please refer to the guide "Working With Expressions". Once you have entered the expression, you must designate that expression as the one you wish to plot.

With the expression at the top of the stack, execute the STEQ operation from the Plot menu. This will pop the expression off of the stack and store it in a symbol called "EQ". At any time, you can execute the RCEQ operation to recall the expression and push it onto the stack.

When you do a plot, the symbol EQ is retrieved from the current directory and the value of that symbol should the expression to plot.

## Adjusting Plot Parameters:

When plotting, there are many pieces of information the calculator needs to know how to plot the expression. It needs to know the bounds of the X/Y coordinate space to plot. It needs to know what symbol in the expression to use for the "X" coordinate. Finally, the calculator needs to know where to position the X and Y axes as well as how much detail to include in the plot.

All of this information is stored in a symbol called PPAR. The value of this symbol is a list of the following items:

1. The bottom left point (plot minimum) in X/Y coordinates expressed as a complex number. The default value is (-6.8, -1.5).
2. The upper right point (plot maximum) in X/Y coordinates expressed as a complex number. The default value is (6.8, 1.5).
3. The independent (X) variable to use when calculating points to plot from the equation. This item should be a symbol and its default value is "constant".
4. The resolution of the plot expressed as a real value. It specifies the number of pixels in the plot view to increment by when calculating the next point. The default value is 1. Increasing this value will result in higher performance since fewer points need to be calculated but less accuracy in the plot.
5. The position of the X/Y axes, specified as a complex number. The default value is (0,0).

If the PPAR symbol does not exist, the calculator will use default values for each of these parameters. Of them, the independent variable is likely the only default parameter you need to change. You can manipulate the list manually (refer to Working With Lists guide for more

information) and store the value you want into a symbol named PPAR. Or you can use the operations in the Plot menu.

First, you will want to set the independent variable using the INDEP operation. This operation takes a symbol from the top of the stack and sets the appropriate item in the PPAR list. If "X" is the symbol representing the X axis in your expression, then push the symbol 'X' onto the stack (refer to the Working With Symbols guide for more information) and run the INDEP operation.

You may want to adjust the visible section of the X/Y coordinate space. You can use the PMIN and PMAX operations to adjust the lower left and the upper right corners of the plot view directly. Or, you can preserve the size of the view and move the center using the CENTR operation. Finally, you can use the *H and the *W to grow or shrink the height or width respectively. These operations take a factor which is greater than 1 to increase that dimension or less than one to shrink it.

If you do not require high precision, you can use the RES operation to reduce the number of points plotted. This can improve the performance of the calculator when working with a plot. You may want to start with a low precision plot to get a sense of what area you are interested in and then once focused there, increase the precision of your plot.

Finally, you can reposition the X and Y axes from their default position at the origin point, (0, 0). Use the AXES operation to specify a new origin point. You might choose to move the axes to give you a reference point on the plot when focused on an area of the coordinate space which is far from (0, 0).

Thankfully, there are more direct ways to manipulate plots which is described later in the Interacting With A Plot section.

## Displaying A Plot:

Once you have specified the expression to plot and setup your plot parameters (or at least the independent variable), you can display the plot simply by executing the DRAW operation from the Plot menu. The stack display will disappear and be replaced with the plot view.

Depending on the position of the X and Y axes and the boundaries of the plot view, a dotted X and/or Y axis may be visible. In the center of the view will be a small plus sign. This is a cursor which allows you to obtain (X,Y) coordinates from the plot and push them onto the stack for later manipulation. Finally, you should see the plot itself being drawn as it is calculated from the left side of the screen towards the right.

At any time you can press the "Attn" key to exit the plot view and return to the usual stack view.

## Interacting With A Plot:

With a plot visible on the calculator, the custom buttons will be replaced with those normally used to edit the command line. There is the insert and delete keys (INS and DEL) and four directional arrow buttons. In plot view, the DEL key does not do anything. But, you can use the arrow buttons to move the cursor which appears as a plus sign. When you move the cursor, a transparent rectangle will appear showing the current (X,Y) coordinates of the cursor. After moving the cursor with the arrow keys, you might want to push the current cursor position onto the stack. Press the INS button to do so. The coordinates are pushed onto the stack as a

complex number where the X position is the real value and the Y position is the imaginary value.

You can also manipulate the cursor directly. Double tap on an area of the plot to move the cursor there. This is helpful for moving the cursor across the view quickly. Drag the cursor with your finger and it will move along with your motions. Double tap on the cursor itself to push the current position onto the stack as a complex number. Often you will find yourself using direct manipulation to move the cursor close to the point of interest and then you might use the arrow buttons for more accuracy.

A common use for values inserted onto the stack from the cursor position is for finding roots, maxima and minima of an expression. When searching for roots, maxima or minima, the calculator needs between 1 and 3 "guesses" where these points lie. So, if you would like to find a root, insert two points on either side of where the plot crosses the X axis. If you would like to find a maxima or minima, insert three points at a local maxima or minima. One point to the left of the max/min value, one point to the right and finally one as close to the max/min as you can get. These points will be excellent input to the SOLVR.

Although you can use operations from the Plot menu to adjust the boundaries of the plot view, it is much more convenient to directly interact with the plot itself. Just drag your finger across the plot view to move the visible coordinate space. Use a pinch gesture to zoom in or out. When you adjust the boundaries of the plot view this way, the plot will be recalculated so you may find the plot disappears for a couple of seconds while it is re-drawn. But this kind of direct manipulation is much easier to use than relying on operations like PMIN or PMAX to find the part of the plot you care about.

Finally, at any time you can rotate your device into landscape orientation. When you do so with a plot visible, the buttons on the calculator will disappear and the plot view will be full screen. This gives you much more resolution and the detail can be very helpful for understanding your expression. Also, pinches and other gestures are easier to perform on a full screen plot than a small plot visible above the calculator buttons. In landscape mode, you can also directly manipulate the cursor. Although you do not have access to the arrow buttons, you can move the cursor to points of interest and double tap it to insert those points onto the stack.

Using these tools, you can quickly find the portion of your expression which is of interest and gather the information you need.

# Halcyon Calc - Working With Programs

Halcyon Calc can combine a series of numbers, strings, lists, expressions and operations together into a simple or complex program. By creating programs, you can define your own operations which can then use in expressions, just as you would use the built-in operations. Programs are only supported on Halcyon Calc. Halcyon Calc Lite does not support programs. Working with programs is described in the following sections:



## Entering Expressions:

A program consists of a series of stack items and operation names between « and » characters. The final » character is optional. If it is left out, one will be added automatically. Any valid stack item can appear inside a program, including complex numbers, strings, lists and even other programs.

By default, the calculator switches to alpha mode when entering a program. Nearly all operations are inserted onto the command line when pressed when in alpha mode. So, to add the EVAL operation in a program, it is as simple as hitting the "EVAL" button.

## Executing Programs:

In the absence of loops and conditional evaluation, a program is evaluated sequentially. Stack items are pushed onto the stack. Operations are executed and operate on items onto the stack, perhaps pushing other items.

There are multiple ways to execute a program. You can use the EVAL or the →NUM to execute a program from the top of the stack. If you store an program into a symbol, pushing that symbol onto the stack will recall the program and execute it.

As an example, we will work with a program which calculates cubes of an input value. If we want to calculate 5 to the power of 3, we can push 5 onto the stack, followed by this program:

```
« 3 ^ »
```

At this point, you can use the EVAL operation to execute the program. During execution, the 3 is pushed onto the stack. Then the ^ (power) operation is executed. The 5 and 3 are popped off the stack and 125 is pushed onto the stack which is $5^3$.

To make this easier, the same program can be pushed onto the stack, followed by the symbol CUBE. Then, using STO, we can store the program into a global symbol called CUBE.

Once that is done, we can execute the program a couple of ways. With 5 on the stack already, we can type the letters to spell "CUBE". Pushing this symbol on the stack will cause the calculator to lookup its value. Finding a program, the calculator will automatically execute the program which will then calculate the cube of 5.

Or, you can hit the "User" button to see the current set of global symbols on the menu buttons. Just hitting the "CUBE" button will execute the program. This is very much like hitting the "SIN" button from the "Trig" menu to calculate the sine of an input value.

There are still more ways to improve the CUBE example and we will return to it in the section about custom operations.

IF operation. This is a compound statement which includes the THEN and END operations and looks like this:

```
« ... IF ... operations ... THEN ... operations ... END ... »
```

When executing a program with an IF operation, execution continues with the stack items and operations up to the THEN operation. When execution reaches the THEN statement, the top of the stack is popped. A real value is expected at the top of the stack or an error will occur. The real value is tested. If it is non-zero, then the if condition is considered to be true, otherwise if the value is zero, it is false. If the condition is true, then the operations between THEN and END are executed. If the conditions are false, then the operations between THEN and END are skipped. Execution always continues after the END operation.

Optionally, you can include a ELSE operation which would look like this:

```
« ... IF ... operations ... THEN ... operations ... ELSE ... operations ... END ... »
```

In this case, if the real value popped at THEN is non-zero (true), then the operations between THEN and ELSE are executed. If the value is zero (false), then the operations between ELSE and END are executed. In both cases, execution continues after the END operation.

Imagine we want to divide two numbers but we want to handle the case where a divide by zero could happen. If a divide by zero would occur, the program pushes the string "Divide by zero". The program will assume that the two numbers to be divided are on the stack already. Here is the program:

```
« DUP IF 0 SAME THEN DROP DROP "Divide by zero" ELSE / END »
```

The program duplicates the top of the stack so it has a copy to test against. It pushes the 0 and then uses the SAME operation to compare the value against 0. If they are the same, the two numbers are dropped from the stack and the string is pushed on. Otherwise, the program executes the divide.

An alternate way to do this is to use the IFERR operation. This operation uses the same structure as IF, with a following THEN and END operation and optionally an ELSE operation. The divide example would now look like this:

```
« IFERR / THEN "Divide by zero" END »
```

With IFERR, execution of the operations between IFERR and THEN occurs normally. But, if an

error is raised by any operation in that sequence, execution jumps from there to the operations following THEN. It may skip one or more operations between IFERR and THEN when the error occurs. Once the operations after the THEN block are executed, execution continues after the END operation normally as though no error occurred. If no-error occurs, execution will jump to an optional ELSE block if it exists or to the operations following END. The IFERR operation is a great way to handle unexpected failures in your programs.

Note that IF and IFERR structures can be nested within each other. You can have an IF/THEN/ELSE/END between the IF and THEN of a different conditional. The key is to keep your IF/THEN/END statements balanced. Similarly for IFERR. Note that if you leave one or more END's out of your program when you enter it, the calculator will add one or more for you to close out any open IF/IFERR statements. If you see END's added which you did not expect, review your program very closely because you likely forgot an END somewhere.

Finally, we can also use the IFT and the IFTE operations as an alternative. Unlike the compound statements used with IF, these operations get the true and an optional false else clause from the stack.

The divide example above could be written like this using IFTE:

```
« DUP 0 SAME « DROP DROP "Divide by zero" » « / » IFTE »
```

In this case, the top of the stack is duplicated and then compared against 0. The result of that compare is still on the stack when two programs are pushed onto the stack. The first program pushed on the stack is executed if the comparison was true (a divide by zero would have occurred). The second program is executed if the comparison was false (no divide by zero). The IFTE operation pops the two programs and the comparison. It then executes the appropriate program based on the value of the comparison.

Either IF or IFT/IFTE can be used for conditional execution. Use whatever you find easiest to work with.

## Flags:

Programs may need a place to store the result of a comparison for later use. The calculator makes available a global 64-bit set of flags. Of them, about half can be used by user programs to store comparison results temporarily. The upper bits hold global calculator state. Changing those bits can change the behaviour of the calculator. The flags changed by a running program continue to hold their values after the program finishes. So, if you change the value of a global configuration it is best to restore the flags when the program completes.

The following table describes the current flag bits which are available:

| Bit(s) | Description |
|---|---|
| 1 - 30 | Available for user programs |
| 31 | Set if LAST is on, default value is set |
| 32 - 34 | Reserved for future use |
| | If set, constants like e will remain in symbolic form. If cleared, constants will |

| | |
|---|---|
| 35 | automatically be replaced with their numeric values. The default value is set. |
| 36 | If set, evaluation of an expression will only lookup each symbol in the expression but when clear, the result of the lookup may itself result in more lookups until a numeric value is found. The default value is set. |
| 37 - 42 | These bits store the current integer word size minus one. So, if the word size is 64, then bits 37 through 42 are all set (ie 63). The default word size is 64. |
| 43 - 44 | These two bits encode the current integer base. With 44 being the leftmost binary digit and 43 the rightmost, 00 is decimal, 01 is octal, 10 is binary and 11 is hexadecimal. The default is decimal. |
| 45 | Set if ML is on, clear otherwise. The default is set. |
| 46 | Set if the previous PUTI or GETI operation wrapped the incoming index. Otherwise, the flag is clear. |
| 47 | Reserved for future use |
| 48 | Set if RDX, is on, clear otherwise. The default value of this bit depends on the international configuration of the iPhone. |
| 49 - 50 | These two bits encode the current number format. With 50 being the leftmost binary digit and 49 the rightmost, 00 is standard mode, 01 is fixed mode, 10 is scientific mode and 11 is engineering mode. The default is standard mode. |
| 51 | When clear, the calculator produces sounds including key clicks. When set, the calculator mutes all sounds. |
| 52 | Reserved for future use |
| 53 - 56 | These bits are used to store the number of digits displayed in the current number format. The default is 0, although in standard format, this value is ignored. |
| 57 - 59 | Reserved for future use |
| 60 | Set if RAD is on, clear if DEG is on. The default is set. |
| 61 - 64 | Reserved for future use |

You can use the FS?, FC? operations to test bits. The FS?C and FC?C operations allow you to test bits and clear them in a single operation. The SF and CF operations can be used to set and clear bits. Finally, you can use the RCLF operation to recall the value of all flags and the STOF operation to set them all in a single operation.

## Loops:

In some programs, you need to execute some operations multiple times. There are many ways to create loops in a program. The START operation is a simple loop which is best used to execute a series of commands a specific number of times.

In this example, we want to calculate the following: 1+2+3+4+5+6+7+8+9+10 We will calculate

this using a START loop:

```
« 0 0 1 10 START 1 + DUP 3 ROLL + SWAP NEXT DROP »
```

This is a bit of a tricky program to follow because it does some stack manipulation to do the job. It pushes a 0 onto the stack which is the starting value for the answer to be calculated. It pushes another 0 onto the stack which will be incremented each time through the loop (ie it will be 1 then 2 etc). It pushes a 1 and a 10 which is the beginning and the end of the START loop. START maintains an internal index which begins at one and by default increments that index by one each time through the loop until the index exceeds the final value (10 in this example).

Between START and NEXT is the body of the loop. In the body, 1 is added to the top of the stack. This is the value which counts from 1 through 10 on the stack. It is duplicated. The first time through the loop, the stack now looks like 0, 1, 1. Then a 3 ROLL is performed. Now the stack looks like 1, 1, 0. The answer we are trying to calculate is at the top of the stack and under that is a copy of the current iteration number. So, we add them to add the current index onto our answer. Finally, we swap these numbers so the current loop count is again on the top of the stack.

Once the loop terminates, we drop the iteration count, leaving 55 which is the right answer. All of this stack manipulation is difficult to get right and often it is useful to have the iteration count handy. For that, use the FOR loop instead. Then, this example looks like this:

```
« 0 1 10 FOR x x + NEXT »
```

This looks much simpler. We still push a zero for a starting value where we will accumulate our answer. We push 1 and 10 for the start and end values for our loop. The FOR operation comes next followed by "x". The FOR operation is always followed by a symbol name and FOR creates a local variable which is valid only through the body of the loop which holds the current iteration number. After this symbol comes the actual body of the loop which continues until the NEXT operation in this example.

The body in this case is x +. The x will result in the value of the local variable being pushed onto the stack (because the symbol name is not in single quotes, it value will be looked up, finding the local value maintained by the FOR loop). Then + is executed adding the current iteration number onto our answer. And then it loops back and does it again.

With both START and FOR, you can use the STEP operation instead of NEXT. With STEP, you can control the value incremented to the loop index. You can add a value greater than 1, fractional values, or even negative values. By using a negative value in a STEP increment, you can iterate from a larger number to a lower value. In some cases, this may be useful for your loops.

Other types of loops may be used when you don't really know how many times you are going to iterate. In these loops, a condition will be tested to see if the loop will continue. The first is the DO, UNTIL and END loop. In this case, the operations between DO and UNTIL are executed repeatedly until the condition evaluated between UNTIL and END evaluates to true. At END, a real value is popped off the stack and the loop continues to execute if the real value is zero.

Our usual addition example now looks like this:

```
« 0 0 DO 1 + DUP 3 ROLL + SWAP UNTIL DUP 10 == END DROP »
```

Again we are back to some serious stack swapping, dropping and rolling. The first zero pushed is where we are accumulating our answer and the second 0 is our iteration count. In the body of the DO loop, we increment the iteration count. Then, we roll the stack, putting our answer on the top of the stack. We add our iteration count to our answer and then swap the current two values on the top of the stack. This puts the iteration count on the top again, our answer below it.

Then, we enter the conditional block. Here we duplicate the top of the stack which is our iteration count and compare it to 10. If the value is 10, then we exit the loop. Otherwise we evaluate the loop again. Once we exit the loop, we drop the iteration count, leaving our answer 55 on the stack.

The other loop type is the WHILE, REPEAT and END loop. With this loop, the condition is evaluated first between WHILE and REPEAT. At REPEAT, a real number is popped off the stack and if it is true (non-zero), then the loop body between REPEAT and END is executed. If it is false, the loop is exited and execution continues after the END. Because the condition is evaluated first, it is possible for the body of the loop to never execute if the condition is false the first time is is evaluated.

Our example now looks like this

```
« 0 0 WHILE DUP 10 < REPEAT 1 + DUP 3 ROLL + SWAP END DROP »
```

The body of the loop between REPEAT and END is the same as what we had in the DO example above between DO and UNTIL. The body of the loop increments the iteration count, adds the current iteration count to the answer and then swaps them just as before. But, before each iteration the condition between WHILE and REPEAT is evaluated. The condition duplicates the iteration count and then tests to see if the iteration count is less than 10. If it is, the loop continues. Once the iteration count reaches 10, the loop exits. The final time through the loop, 9 is compared to 10 and the condition is true. So, the body of the loop is evaluated one last time. The 9 is incremented to 10 and 10 is added to the answer. When the condition is evaluated again, 10 is compared to 10 and the condition is false so the loop exits. As before, the iteration count is dropped, leaving the answer 55 at the top of the stack.

In this example, the FOR loop is the most natural solution. But, these different forms of loops are available and you may find some more appropriate in different situations.

## Creating Custom Operations:

Halcyon Calc has a large suite of built-in operations but you may find a specific operation missing which you use frequently. Above, we created a program to calculate cubes. Wouldn't it be great if you could use that program in your own expressions like 'CUBE(5)' or even better 'CUBE(SIN(X + Y))'?

To do this, we will use the → operation. If it is used like this:

```
« → symbol1 ... symboln « anotherProgram » »
```

or instead of a program, you can use an expression like this:

```
« → symbol1 ... symboln 'expression' »
```

Then when stored in a global variable, this program becomes as flexible as the other built-in operations in the calculator. When executed, the list of "n" symbols represents the arguments the operation takes. It can take these arguments from the stack or it can take them from an argument list in brackets. The values for these symbols, regardless of where they came from, are then set in local variables when the final program or expression is evaluated.

For our CUBE program, we can do this:

```
« → x « x 3 ^ » »
```

Then, store this program in a global variable called CUBE. What this does is takes one argument from the stack or from an argument list in brackets and sets the local variable "x" to this value. The inner program is executed with that local variable set. It pushes the value of x on the stack, followed by a 3 and then executes the power operation. This calculates the cube of the incoming argument.

Alternatively, you can use an expression instead of an embedded program:

```
« → x 'x^3' »
```

Again, store this in a global variable called CUBE to use it. Either way, the CUBE program becomes indistinguishable from built-in operations like SIN. You can push 5 onto the stack and hit the "CUBE" button from the User menu to calculate the cube of 5. You can push 'CUBE(5)' onto the stack and then evaluate it to get 125. Your custom operation can appear in expressions which evaluate with the solver, search for roots using the ROOT operation or just about anything else.

## Debugging Programs:

Often, you may find your program doesn't do what you expect. There are several things you can do to solve the problem. First, it is best to break programs up into smaller components. Better than having a large, complicated program is to create a series of smaller programs which call each other to get a larger job done. The art is finding the way to break it up appropriately. But, with a program split into smaller parts, you can debug a single part at a time to try and find the problem.

Given a program you want to debug is on the top of the stack, you can use ▪Edit to edit the program. At an appropriate place where you want to see what is going on, you can insert the HALT operation and then execute the program. When the HALT operation is reached, the program stops executing. You can see the state of the stack at this point.

Perhaps the problem is obvious. If you think you can fix things in place, you can change the stack to what it should look like using a series of operations. When you are ready, you can use CONT to resume execution.

If you don't see the problem yet, you can single step the program using the SST operation. The program will execute a single item at a time, pushing items onto the stack or executing a single operation. As you step through the program, the problem may become apparent.

Once you know what the problem is, you can use ▪Edit to edit the program and fix it. Don't forget to store it since most programs end up stored in global variables.

If after this you still can't figure out what the problem is, post a question to our forum and we will see if we can help out.

## Manipulating Programs:

There are many resources which you can use when working with programs:

- The Control Menu, Branch Menu and Test Menu have several useful operations which you can use within your programs or for debugging your programs.
- A list of operations which take programs can be found here.
- A list of operations which produce programs can be found here.

# Halcyon Calc - Built-In Units

The following table lists all of the build-in units which you can use with the CONVERT operation.

| Unit | Name | Unit Type | Value |
|---|---|---|---|
| a | Are | Area | 100 $m^2$ |
| A | Ampere | Electric current | 1 A |
| acre | Acre | Area | 4046.87260987 $m^2$ |
| arcmin | Minute of arc | Plane angle | 4.62962962963E-5 |
| arcs | Second of arc | Plane angle | 7.71604938272E-7 |
| atm | Atmosphere | Pressure | 101325 kg / m * $s^2$ |
| au | Astronomical unit | Length | 149597900000 m |
| A° | Angstrom | Length | 0.0000000001 m |
| b | Barn | Area | 1.0E-28 $m^2$ |
| bar | Bar | Pressure | 100000 kg / m * $s^2$ |
| bbl | Barrel of oil | Volume | 0.158987294928 $m^3$ |
| Bq | Becquerel | Activity | 1 1 / s |
| Btu | International Table Btu | Energy | 1055.05585262 kg * $m^2$ / $s^2$ |
| bu | Bushel | Volume | 0.03523907 $m^3$ |
| c | Speed of light | Velocity | 299792458 m / s |
| C | Coulomb | Electric charge | 1 A * s |
| cal | International Table calorie | Energy | 4.1868 kg * $m^2$ / $s^2$ |
| cd | Candela | Luminous intensity | 1 cd |
| chain | Chain | Length | 20.1168402337 m |
| Ci | Curie | Activity | 3.7E10 1 / s |
| ct | Carat | Mass | 0.0002 kg |
| cu | US cup | Volume | 2.365882365E-4 $m^3$ |
| d | Day | Time | 86400 s |
| dyn | Dyne | Force | 0.00001 kg * m / $s^2$ |
| erg | Erg | Energy | 0.0000001 kg * $m^2$ / $s^2$ |
| eV | Electron volt | Energy | 1.60219E-19 kg * $m^2$ / $s^2$ |
| F | Farad | Capacitance | 1 $A^2$ * $s^4$ / kg * $m^2$ |

| | | | |
|---|---|---|---|
| fath | Fathom | Length | 1.82880365761 m |
| fbm | Board foot | Colume | 0.002359737216 $m^3$ |
| fc | Footcandle | Luminance | 0.856564774909 cd / $m^2$ |
| Fdy | Faraday | Electric charge | 96487 A * s |
| fermi | Fermi | Length | 1.0E-15 m |
| flam | Footlambert | Luminance | 3.42625909964 cd / $m^2$ |
| ft | International foot | Length | 0.3048 m |
| ftUS | Survey foot | Length | 0.304800609601 m |
| g | Gram | Mass | 0.001 kg |
| ga | Standard freefall | Acceleration | 9.80665 m / $s^2$ |
| gal | US gallon | Volume | 0.003785411784 $m^3$ |
| galC | Canadian gallon | Volume | 0.00454609 $m^3$ |
| galUK | UK gallon | Volume | 0.004546092 $m^3$ |
| gf | Gram-force | Force | 0.00980665 kg * m / $s^2$ |
| grad | Grade | Plane angle | 0.0025 |
| grain | Grain | Mass | 0.00006479891 kg |
| Gy | Gray | Absorbed dose | 1 $m^2$ / $s^2$ |
| h | Hour | Time | 3600 s |
| H | Henry | Inductance | 1 kg * $m^2$ / $A^2$ * $s^2$ |
| hp | Horsepower | Power | 745.699871582 kg * $m^2$ / $s^3$ |
| Hz | Hertz | Frequency | 1 1 / s |
| in | Inch | Length | 0.0254 m |
| inHg | Inches of mercury | Pressure | 3386.38815789 kg / m * $s^2$ |
| inH2O | Inches of water | Pressure | 248.84 kg / m * $s^2$ |
| J | Joule | Energy | 1 kg * $m^2$ / $s^2$ |
| kip | Kilopound-force | Force | 4448.22161526 kg * m / $s^2$ |
| knot | Knot | Speed | 0.514444444444 m / s |
| kph | Kilometer per hour | Speed | 0.277777777778 m / s |
| l | Liter | Volume | 0.001 $m^3$ |
| lam | Lambert | Luminance | 3183.09886184 cd / $m^2$ |

| | | | |
|---|---|---|---|
| lb | Avoirdupois pound | Mass | 0.45359237 kg |
| lbf | Pound-force | Force | 4.44822161526 kg * m / $s^2$ |
| lbt | Troy lb | Mass | 0.3732417 kg |
| lm | Lumen | Luminance flux | 7.95774715459E-2 cd |
| lx | Lux | Illuminance | 7.95774715459E-2 cd / $m^2$ |
| lyr | Light year | Length | 9.46052840488E15 m |
| m | Meter | Length | 1 m |
| mho | Mho | Electrical conductance | 1 $A^2$ * $s^3$ / kg * $m^2$ |
| mi | International mile | Length | 1609.344 m |
| mil | Mil | Length | 0.0000254 m |
| min | Minute | Time | 60 s |
| miUS | US statute mile | Length | 1609.34721869 m |
| mmHg | Millimeter of mercury | Pressure | 133.322368421 kg / m * $s^2$ |
| mol | Mole | Amount of substance | 1 mol |
| mph | Miles per hour | Speed | 0.44704 m / s |
| N | Newton | Force | 1 kg * m / $s^2$ |
| nmi | Nautical mile | Length | 1852 m |
| ohm | Ohm | Electrical resistance | 1 kg * $m^2$ / $A^2$ * $s^3$ |
| oz | Ounce | Mass | 0.028349523125 kg |
| ozfl | US fluid ounce | Volume | 2.95735295625E-5 $m^3$ |
| ozt | Troy ounce | Mass | 0.031103475 kg |
| ozUK | UK fluid ounce | Volume | 0.000028413075 $m^3$ |
| P | Poise | Dynamic viscosity | 0.1 kg / m * s |
| Pa | Pascal | Pressure | 1 kg / m * $s^2$ |
| pc | Parsec | Length | 3.08567818585E16 m |
| pdl | Poundal | Force | 0.138254954376 kg * m / $s^2$ |
| ph | Phot | Luminance | 795.774715459 cd / $m^2$ |
| pk | Peck | Volume | 0.0088097675 $m^3$ |
| psi | Pounds per square inch | Pressure | 6894.75729317 kg / m * $s^2$ |
| pt | Pint | Volume | 0.000473176473 $m^3$ |

| | | | |
|---|---|---|---|
| qt | Quart | Volume | 0.000946352946 $m^3$ |
| r | Radian | Plane angle | 0.159154943092 |
| R | Roentgen | Radiation exposure | 0.000258 A * s / kg |
| rad | Rad | Absorbed dose | 0.01 $m^2$ / $s^2$ |
| rd | Rod | Length | 5.02921005842 m |
| rem | Rem | Dose equivalent | 0.01 $m^2$ / $s^2$ |
| s | Second | Time | 1 s |
| S | Siemens | Electric conductance | 1 $A^2$ * $s^3$ / kg * $m^2$ |
| sb | Stib | Luminance | 10000 cd / $m^2$ |
| slug | Slug | Mass | 14.5939029372 kg |
| sr | Steradian | Solid angle | 7.95774715459E-2 |
| st | Stere | Volume | 1 $m^3$ |
| St | Stokes | Kinematic viscosity | 0.0001 $m^2$ / s |
| Sv | Sievert | Dose equivalent | 1 $m^2$ / $s^2$ |
| t | Metric ton | Mass | 1000 kg |
| T | Tesla | Magnetic induction | 1 kg / A * $s^2$ |
| tbsp | Tablespoon | Volume | 1.47867647813E-5 $m^3$ |
| therm | EEC therm | Energy | 105506000 kg * $m^2$ / $s^2$ |
| ton | Short ton | Mass | 907.18474 kg |
| tonUK | Long ton | Mass | 1016.0469088 kg |
| torr | Torr | Pressure | 133.322368421 kg / m * $s^2$ |
| tsp | Teaspoon | Volume | 4.92892159375E-6 $m^3$ |
| u | Unified atomic mass | Mass | 1.66058E-27 kg |
| V | Volt | Electric potential | 1 kg * $m^2$ / A * $s^3$ |
| W | Watt | Power | 1 kg * $m^2$ / $s^3$ |
| Wb | Weber | Magnetic flux | 1 kg * $m^2$ / A * $s^2$ |
| yd | International yard | Length | 0.9144 m |
| yr | Year | Time | 31556925.9747 s |
| ° | Degree | Angle | 2.777777777778E-3 |
| °C | Degree Celsius | Temperature | 1 °K |

| | | | |
|---|---|---|---|
| °F | Degree Fahrenheit | Temperature | 0.555555555556 °K |
| °K | Degree Kelvin | Temperature | 1 °K |
| °R | Degree Rankine | Temperature | 0.555555555556 °K |
| µ | Micron | Length | 0.000001 m |
| ? | User quantity | | 1 ? |
| 1 | Dimensionless unit | | 1 |

# Operation Reference:

Documentation for each operation provided in Halcyon Calc is provided in reference form on the remaining pages of this document.

# %

**Calculator Key:** ■%

**Member Of Menu:** None
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation takes two real arguments, calculates the product of them and divides that product by 100.

# %CH

**Calculator Key:** ■%CH

**Member Of Menu:** None
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

Given real values x and y pushed onto the stack in that order, this operation calculates 100 * (y - x) / x. Note that if the first value is 0, an infinite result error is displayed.

# %T

**Member Of Menu:** Real
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

Given $Real_1$ and $Real_2$ from the stack, this function computes 100 * $Real_2$ / $Real_1$.

# *H

**Member Of Menu:** Plot
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Real_1$ → 

This operation takes a real number from the top of the stack. It then adjusts the plot minimum (see PMIN) and the plot maximum (see PMAX) so that the current center of the plot is preserved but the height of the plot is multiplied by the factor specified. A value greater than 1 will result in the plot height increasing while a value less than 1 will decrease the plot height. The current Y values for the plot minimum and the plot maximum will change accordingly but the X values will be preserved.

If PPAR exists, its values are read, the new plot minimum and maximum is calculated and then PPAR is stored, overwriting its old values. If PPAR does not exist, then default values are used, the plot view height is recalculated and then PPAR is stored.

# *W

**Member Of Menu:** Plot
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Real_1$ → 

This operation takes a real number from the top of the stack. It then adjusts the plot minimum (see PMIN) and the plot maximum (see PMAX) so that the current center of the plot is preserved but the width of the plot is multiplied by the factor specified. A value greater than 1 will result in the plot width increasing while a value less than 1 will decrease the plot width. The current X values for the plot minimum and the plot maximum will change accordingly but the Y values will be preserved.

If PPAR exists, its values are read, the new plot minimum and maximum is calculated and then PPAR is stored, overwriting its old values. If PPAR does not exist, then default values are used, the plot view width is recalculated and then PPAR is stored.

# +

**Calculator Key:** +

**Member Of Menu:** None
**Argument Types:** Real
Complex
Integer
List
String
Symbol
Expression
Array
**Result Type(s):** Real
Complex
Integer
List
String
Expression
Array
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Complex_1$ | $Complex_2$ | → | $Complex_3$ |
| $Real_1$ | $Complex_2$ | → | $Complex_3$ |
| $Complex_1$ | $Real_2$ | → | $Complex_3$ |
| $Integer_1$ | $Integer_2$ | → | $Integer_3$ |
| $Real_1$ | $Integer_2$ | → | $Integer_3$ |
| $Integer_1$ | $Real_2$ | → | $Integer_3$ |
| { $List_1$ } | { $List_2$ } | → | { $List_1$ $List_2$ } |
| { $List_1$ } | $Item_2$ | → | { $List_1$ $Item_2$ } |
| $Item_1$ | { $List_2$ } | → | { $Item_1$ $List_2$ } |
| $String_1$ | $String_2$ | → | $String_3$ |
| $Array_1$ | $Array_2$ | → | $Array_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

The add operation will take its two numerical operands and produce the sum as its result. It operates on reals, complex and integer values. Also, you can combine reals with complex values and reals with integer values. The result will be a complex or integer value respectively.

Two lists can be added and the result is a concatenation of the two lists. Also, you can add any item on the stack to a list. If the item is put on the stack first, then it will be prepended to the list. Otherwise, it will be appended to the list.

Finally, you can add strings to each other. The result is a concatenation of the two string values.

Matrices and vectors can be added together. A vector cannot be added to a matrix and vice versa. The number of rows and columns in the matrices must match. The vectors or matrices being added can be real, complex or a mixture of real and complex. The result will be a a vector or matrix of the same dimension as the input values with each corresponding value being the sum of the input values at that position.

# -

**Calculator Key:** -

**Member Of Menu:** None
**Argument Types:** Real
Complex
Integer
Symbol
Expression
Array
**Result Type(s):** Real
Complex
Integer
Expression
Array
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Complex_1$ | $Complex_2$ | → | $Complex_3$ |
| $Real_1$ | $Complex_2$ | → | $Complex_3$ |
| $Complex_1$ | $Real_2$ | → | $Complex_3$ |
| $Integer_1$ | $Integer_2$ | → | $Integer_3$ |
| $Real_1$ | $Integer_2$ | → | $Integer_3$ |
| $Integer_1$ | $Real_2$ | → | $Integer_3$ |
| $Array_1$ | $Array_2$ | → | $Array_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

The subtract operation will take its two numerical operands and produce the difference as its result. It operates on reals, complex and integer values. Also, you can combine reals with complex values and reals with integer values. The result will be a complex or integer value respectively.

Matrices and vectors can be subtracted. A vector cannot be subtracted from a matrix and vice versa. The number of rows and columns in the matrices must match. The vectors or matrices being subtracted can be real, complex or a mixture of real and complex. The result will be a a vector or matrix of the same dimension as the input values with each corresponding value being the difference of the input values at that position.

# <

**Calculator Key:** ▪<

**Member Of Menu:** None
**Argument Types:** Real
Integer
String
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Integer_1$ | $Integer_2$ | → | $Real_3$ |
| $String_1$ | $String_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation takes two real, integer or string values and produces a 1 if the first value is less than the second, 0 otherwise. The result is always a real, even if the incoming arguments are integers or strings.

# =

**Calculator Key:** =

**Member Of Menu:** None
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Expression_3$ |
| $Real_1$ | $Complex_2$ | → | $Expression_3$ |
| $Complex_1$ | $Real_2$ | → | $Expression_3$ |
| $Complex_1$ | $Complex_2$ | → | $Expression_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation always produces an expression which describes a relationship between its left and right side. Unlike the == operator which determines whether one value equals another, this operation is like a mathematical relationship. A good example of this kind of expression is the standard equation for a line, y = mx + b. Assuming that m and b are constant values, the equation creates a relationship between x an y. Also, the equation can be manipulated to produce equivalent equations like y - b = mx.

Similarly, expressions which contain an equals operation can also be manipulated in the same way on the calculator. Assume that 3 is added to the expression "left=right". This would produce the equation "left+3=right+3". The operation is applied to both sides of the equation to balance it. This works for addition and all other operations which take expressions.

Also, assume there are two equations being added together, "A=B" and "C=D". This will result in "A+B=C+D". Again, this works with operations like add and all others which operate on expressions.

# ==

**Member Of Menu:** Test
**Argument Types:** Real
Complex
Integer
List
String
Symbol
Expression
Program
Array
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Real_1$ | $Complex_2$ | → | $Real_3$ |
| $Complex_1$ | $Real_2$ | → | $Real_3$ |
| $Complex_1$ | $Complex_2$ | → | $Real_3$ |
| $Integer_1$ | $Integer_2$ | → | $Real_3$ |
| $Real_1$ | $Integer_2$ | → | $Real_3$ |
| $Integer_1$ | $Real_2$ | → | $Real_3$ |
| $List_1$ | $List_2$ | → | $Real_3$ |
| $String_1$ | $String_2$ | → | $Real_3$ |
| $Array_1$ | $Array_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |
| $Program_1$ | $Program_2$ | → | $Real_3$ |

This operation produces a 1 if the two arguments are equal, a 0 otherwise. It can operate on real, complex, integer, list or string values. It also can compare real and complex numbers and real and integer numbers. Note that a vector is equal if it has the same number of values and those values are the same. Similarly, a matrix is equal if it has the same number of rows and columns and each value is the same.

Note that this operation does not appear on a calculator key directly but it can be used by pressing the "=" key twice.

# >

**Calculator Key:** ■>

**Member Of Menu:** None
**Argument Types:** Real
Integer
String
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Integer_1$ | $Integer_2$ | → | $Real_3$ |
| $String_1$ | $String_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation takes two real, integer or string values and produces a 1 if the first value is greater than the second, 0 otherwise. The result is always a real, even if the incoming arguments are integers or strings.

# ABORT

**Member Of Menu:** Control
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

The ABORT operation can be used within an executing program to halt execution and discard execution state. Because execution state is discarded, the program cannot be continued or single-stepped. If this operation is used outside of an executing program, the execution state of the most recently HALT-ed program is discarded.

# ABS

**Member Of Menu:** Complex
Real
Array
**Argument Types:** Real
Complex
Symbol
Expression
Array
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Real_2$ |
| $Array_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function returns the absolute value of its input argument. For a real number input, the result is always a positive real number with the same magnitude. So, -5 will become 5 when passed through this function.

For a complex input, the length of a vector rooted at the origin to the coordinate on the real and imaginary axis is the result of this function.

For complex or real matrices and vectors, the result is the Frobenius norm of the array. This is equal to the square root of the sum of the squares of the absolute values of each element of the array.

# ACOS

**Member Of Menu:** Trig
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Real_1$ | → | $Complex_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the inverse cosine function of its argument. Note that the result depends on the whether the DEG (degree) or RAD (radians) mode is on. For real valued results, the result is expressed as an angle in degrees if DEG is on. For real valued results, the argument is expressed as an angle in radians if RAD is on. For complex arguments, the result is always expressed in radians.

The result lies in the range of 0 to 180 degrees if operating in degree mode, or in the 0 to π range if operating in radian mode.

# ACOSH

**Member Of Menu:** Logs
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Real_1$ | → | $Complex_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the inverse hyperbolic cosine function of its input argument. It takes real or complex input values. Note that if its real argument is less than 1, the result will be complex.

# ALOG

**Member Of Menu:** Logs
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the reverse base 10 logarithm of its input argument. Assuming that the input argument is x, this is equivalent to $10^x$. Real and complex numbers are valid inputs to this operation.

# AND

**Member Of Menu:** Binary
Test
**Argument Types:** Real
Integer
Symbol
Expression
**Result Type(s):** Real
Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Integer_1$ | $Integer_2$ | → | $Integer_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation performs a binary and operation on its two integer arguments and returns the integer result. If the input arguments are real values, it returns a 1 if both inputs are non-zero, otherwise it returns 0.

Note that when used in an expression on symbols x and y for example, it would look like 'x AND y'.

# ARG

**Member Of Menu:** Trig
Complex
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function takes a complex argument and returns the angle given the real and imaginary components of that complex value. That angle may be expressed in degrees or radians depending on the mode of the calculator (see RAD and DEG for more information about these modes). If this function is passed a positive real value, 0 is returned. If this function is passed a negative real value, 180 degrees or π radians is returned (depending on whether operating in degrees or radians mode).

# ARRY→

**Member Of Menu:** Array
**Argument Types:** Array
**Result Type(s):** Real
Complex
List
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | | | |
|---|---|---|---|---|---|
| [ $Real_1$ ... $Real_n$ ] | → | $Real_1$ | ... | $Real_n$ | $List_{n+1}$ |
| [ $Complex_1$ ... $Complex_n$ ] | → | $Complex_1$ | ... | $Complex_n$ | $List_{n+1}$ |
| [[ $Real_1$ ... $Real_n$ ]] | → | $Real_1$ | ... | $Real_n$ | $List_{n+1}$ |
| [[ $Complex_1$ ... $Complex_n$ ]] | → | $Complex_1$ | ... | $Complex_n$ | $List_{n+1}$ |

This operation takes a vector or matrix and pushes each value within that array followed by a list describing the dimensions of the array. For a vector, the list will have a single real value which is the number of values in the vector. For a matrix, the list will have two real values which are the number of rows and the number of columns in the matrix.

# ASIN

**Member Of Menu:** Trig
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Real_1$ | → | $Complex_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the inverse sine function of its argument. Note that the result depends on the whether the DEG (degree) or RAD (radians) mode is on. For real valued results, the result is expressed as an angle in degrees if DEG is on. For real valued results, the argument is expressed as an angle in radians if RAD is on. For complex arguments, the result is always expressed in radians.

The result lies in the range of -90 to 90 degrees if operating in degree mode, or in the $-\pi/2$ to $\pi/2$ range if operating in radian mode.

# ASINH

**Member Of Menu:** Logs
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the inverse hyperbolic sine function of its input argument. It takes real or complex input values.

# ASR

**Member Of Menu:** Binary
**Argument Types:** Integer
Symbol
Expression
**Result Type(s):** Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Integer_1$ | → | $Integer_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes an integer from the stack, shifts each bit one position to the right. The lower bit is lost. If the old upper bit is a zero then a zero is shifted into the upper bit position (which depends on the current word size of the calculator) otherwise a one is shifted into the upper bit position. Assuming a 4 bit word size, then the binary number 1001 will become 1100 after this operation.

This operation stands for arithmetic shift right.

# ATAN

**Member Of Menu:** Trig
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the inverse tangent function of its argument. Note that the result depends on the whether the DEG (degree) or RAD (radians) mode is on. For real valued results, the result is expressed as an angle in degrees if DEG is on. For real valued results, the argument is expressed as an angle in radians if RAD is on. For complex arguments, the result is always expressed in radians.

The result lies in the range of -90 to 90 degrees if operating in degree mode, or in the $-\pi/2$ to $\pi/2$ range if operating in radian mode.

# ATANH

**Member Of Menu:** Logs
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Real_1$ | → | $Complex_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the hyperbolic sine function of its input argument. It takes real or complex input values. Note that the function produces an infinite result error if the input argument is 1 or -1. For real inputs between -1 and 1, the result will be real but for all other inputs, the result will be complex.

# AXES

**Member Of Menu:** Plot
**Argument Types:** Complex
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Complex_1$ →

This operation takes a complex number from the top of the stack and interprets that complex number as X and Y coordinates. It then uses that point as the location where the X and Y axes should intersect for a subsequent plot and stores that information in the plot parameters, found in a variable called PPAR in the current directory. The existing axes which may be specified in an existing PPAR variable will be changed and then PPAR is stored, replacing the old value. If PPAR did not exist prior to executing these operations, a new PPAR variable is created. Its contents will be defaults except for the plot minimum which is specified from the stack.

The default position for the axes is (0,0) but you may want to move the axes to give you some guidance where you are in the plot when you are not plotting near the origin.

# BIN

**Member Of Menu:** Binary
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation puts the calculator in binary mode. Integer values are displayed in base 2. The integer 100 binary will be shown as "# 100b". The "#" at the start indicates that the number is an integer and the trailing "b" indicates the number is displayed in binary mode.

When entering integers, the final character normally indicates what base to use when reading that value. The characters are d for decimal, h for hexadecimal, o for octal and b for binary. Omitting that character in binary mode will result in the calculator assuming the value should be interpreted as a binary value.

# B→R

**Member Of Menu:** Binary
**Argument Types:** Integer
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $\text{Integer}_1$ | → | $\text{Real}_2$ |
| $\text{Symbol}_1$ | → | $\text{Expression}_2$ |
| $\text{Expression}_1$ | → | $\text{Expression}_2$ |

This operation takes an integer value an converts it to the equivalent real value.

# CEIL

**Member Of Menu:** Real
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

Given a real valued input, this function returns the smallest integer which is greater than or equal to the input value.

# CENTR

**Member Of Menu:** Plot
**Argument Types:** Complex
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Complex_1$ | → |

This operation takes a complex number from the top of the stack and interprets that complex number as X and Y coordinates. It then adjusts the plot minimum (see PMIN) and the plot maximum (see PMAX) so that the point specified is at the center of the plot. The current width and height of the plot given the current PPAR values is preserved but the actual plot minimum and maximum is changed.

If PPAR exists, its values are read, the new plot minimum and maximum is calculated and then PPAR is stored, overwriting its old values. If PPAR does not exist, then default values are used, the plot view repositioned with the new center point and then PPAR is stored.

# CF

**Member Of Menu:** Test
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Real_1$ →

This operation takes a real value between 1 and 64 and sets the associated flag bit to zero. See this page for information about the calculator flags.

# CHR

**Member Of Menu:** String
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** String
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $String_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes a real value and maps it to a single character which it returns as a string value. The real value is rounded. If it is below 0, it is treated as 0. If the value is above 256 or more, it uses the modulus of 256 (so 256 is treated as 0, 257 is treated as 1 etc). The following table describes the mapping:

| Real Value | Character |
|---|---|
| 0 - 31 | ▪ |
| 32 | <space> |
| 33 | ! |
| 34 | \ |
| 35 | # |
| 36 | $ |
| 37 | % |
| 38 | & |
| 39 | ' |
| 40 | ( |
| 41 | ) |
| 42 | * |
| 43 | + |
| 44 | , |
| 45 | - |
| 46 | . |
| | |

| | |
|---|---|
| 47 | / |
| 48 | 0 |
| 49 | 1 |
| 50 | 2 |
| 51 | 3 |
| 52 | 4 |
| 53 | 5 |
| 54 | 6 |
| 55 | 7 |
| 56 | 8 |
| 57 | 9 |
| 58 | : |
| 59 | ; |
| 60 | < |
| 61 | = |
| 62 | > |
| 63 | ? |
| 64 | @ |
| 65 | A |
| 66 | B |
| 67 | C |
| 68 | D |
| 69 | E |
| 70 | F |
| 71 | G |
| 72 | H |
| 73 | I |
| 74 | J |
| 75 | K |
| 76 | L |
| 77 | M |
| 78 | N |

| | |
|---|---|
| 79 | O |
| 80 | P |
| 81 | Q |
| 82 | R |
| 83 | S |
| 84 | T |
| 85 | U |
| 86 | V |
| 87 | W |
| 88 | X |
| 89 | Y |
| 90 | Z |
| 91 | [ |
| 92 | \ |
| 93 | ] |
| 94 | ^ |
| 95 | _ |
| 96 | ` |
| 97 | a |
| 98 | b |
| 99 | c |
| 100 | d |
| 101 | e |
| 102 | f |
| 103 | g |
| 104 | h |
| 105 | i |
| 106 | j |
| 107 | k |
| 108 | l |
| 109 | m |
| | |

| | |
|---|---|
| 110 | n |
| 111 | o |
| 112 | p |
| 113 | q |
| 114 | r |
| 115 | s |
| 116 | t |
| 117 | u |
| 118 | v |
| 119 | w |
| 120 | x |
| 121 | y |
| 122 | z |
| 123 | { |
| 124 | \| |
| 125 | } |
| 126 | ~ |
| 127 | ▦ |
| 128 | <space> |
| 129 | ÷ |
| 130 | × |
| 131 | √ |
| 132 | ∫ |
| 133 | ∑ |
| 134 | ▶ |
| 135 | π |
| 136 | ∂ |
| 137 | ≤ |
| 138 | ≥ |
| 139 | ≠ |
| 140 | ∝ |
| | |

| | |
|---|---|
| 141 | → |
| 142 | ← |
| 143 | µ |
| 144 | ■ |
| 145 | ° |
| 146 | ≪ |
| 147 | ≫ |
| 148 - 255 | ■ |

# CLEAR

**Calculator Key:** ▪Clear

**Member Of Menu:** None
**Argument Types:** Any
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Item_1$ | ... | $Item_n$ | → |

This operation removes all items from the stack, leaving an empty stack.

# CLUSR

**Member Of Menu:** Memory
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation is used to clear all variables and empty directories in the current directory. Any directories which are not empty will remain after executing this operation.

Because of the potential impact of inadvertently executing this operation, pressing the CLUSR button will only put "CLUSR" into the calculators entry. You must then press "Enter" to actually execute the operation.

# CL∑

**Member Of Menu:** Stat
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

→

This operation deletes a variable called ∑DAT which normally contains a real matrix of samples used by other statistics operations.

# CMD

**Member Of Menu:** Mode
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation enables or disables the Command functionality on the calculator which allows you to recall recently entered values on the stack and edit them as a new entry string. See the Command documentation for more information.

# CNRM

**Member Of Menu:** Array
**Argument Types:** Array
**Result Type(s):** Real
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Array_1$ | → | $Real_2$ |

This operation calculates the column norm or one-norm of the input vector or matrix.

# COLCT

**Member Of Menu:** Algebra
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Symbol
Expression
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Symbol_2$ |
| $Expression_1$ | → | $Expression_2$ |

The COLCT operation collects terms and factors in order to simplify an expression. It does this by doing the following things:

- It evaluates any parts of the expression which have numerical arguments.
- It collects together real and complex terms across addition and subtraction. So, if you have an expression like '10+X-(1,2)', it will return '(9,-2)+X'.
- It collects together real and complex factors across multiplication and division. So, if you have an expression like '(2,4)*X/2', it will return '(1,2)*X'.
- It reorders the components of a sequence of addition and subtraction terms, combining like terms together as much as possible. This means an expression like 'Y+Z+4*X-Z+10+2*Y-X' will become '10+3*X+3*Y'.
- It reorders the components of a sequence of multiplication and division factors, combining like factors together as much as possible. This means an expression like 'SQ(X)*Y^3*Z^A/X*Y^-4*Z' will become 'X*Z^(1+A)/Y'.

If the COLCT operation encounters an operation it cannot collect, it will try to collect the arguments to that operation. So, 'SIN(10+X+2+X)' will become 'SIN(12+2*X)'.

Similarly, each factor or term can be an expression that collect can't necessarily operate on but it will still collect them. So, 'SIN(X)+2*SIN(X)' will become '3*SIN(X)'.

The COLCT operation attempts to perform all possible collect opportunities across the whole expression in a single call. Re-executing COLCT a second time should never result in a further "collected" expression. This differs from EXPAN which will perform a single expansion with every execution until no further expansions are possible.

# COL∑

**Member Of Menu:** Stat
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | $Real_2$ | → |

This operation is used to designate the dependent and independent column in the statistics data. $Real_1$ specifies the column which contains the independent variable and $Real_2$ specifies the column which contain the dependent variable.

This information is used in other operations like CORR, COV and LR. The information is stored as a list in a variable called ∑PAR. If the ∑PAR variable does not exist, it will be created. A list with value { $Real_1$, $Real_2$, 0, 0 } will be stored in the list. If ∑PAR does exist, it must contain a list with four real values. The first two values in the list are replaced with $Real_1$ and $Real_2$. The third and fourth real values in the list are preserved.

There is no validation that the real values pulled from the stack and stored in ∑PAR are valid values. Values should be whole numbers greater than or equal to one and less than or equal to the number of columns in the ∑DAT real matrix. Invalid values will be detected when the ∑PAR variable is used in the CORR, COV or LR operations.

# COMB

**Member Of Menu:** Stat
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation calculates the number of combinations given $Real_1$ items taken $Real_2$ at a time.

# CON

**Member Of Menu:** Array
**Argument Types:** Real
Complex
List
Symbol
Array
**Result Type(s):** None
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $List_1$ | $Real_2$ | → | $Array_3$ |
| $List_1$ | $Complex_2$ | → | $Array_3$ |
| $Array_1$ | $Real_2$ | → | $Array_3$ |
| $Array_1$ | $Complex_2$ | → | $Array_3$ |
| $Symbol_1$ | $Real_2$ | → | |
| $Symbol_1$ | $Complex_2$ | → | |

This operation is used to create an array of constant values. The resulting array will have all values set to the real or complex argument provided on input. If the first argument is a list, the list contains one or two real values which specifies the dimension of the resulting array. If the list has a single real value, the result will be a vector. If the list has two values, it specifies the number of rows and columns in a resulting matrix.

If the first argument is a vector or matrix, then the result will also be a vector or matrix of the same dimensions. If the second argument is complex, then the vector or matrix provided must also be complex.

Finally, if the first argument is a symbol, then the value at that symbol must be a vector or matrix. If the second argument is complex, then the vector or matrix value of that symbol must also be complex. The value of the symbol is updated to be a constant vector or matrix with value specified by the second argument.

# CONJ

**Member Of Menu:** Complex
Array
**Argument Types:** Real
Complex
Symbol
Expression
Array
**Result Type(s):** Real
Complex
Expression
Array
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_1$ |
| ($Real_1$, $Real_2$) | → | ($Real_1$, -$Real_2$) |
| $Array_1$ | → | $Array_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function takes a complex value and returns its conjugate value. A conjugate value has the same real component but its imaginary component is multiplied by -1. For a real input, that same real value is returned by this function.

This operation also works with matrices and vectors. If the input matrix or vector is real valued, the result of the operation is the same as its input. If the matrix or vector is complex valued, then the result is a matrix or vector of same dimension as the input with each value set to its conjugate.

# CONT

**Calculator Key:** ▪Cont

**Member Of Menu:** None
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

The CONT operation will resume a HALT-ed program. By placing a HALT in your program at an opportune location, you can inspect the state of the stack, make some changes if things aren't quite right and then continue by using this button.

# CONVERT

**Calculator Key:** ▪Convert

**Member Of Menu:** None
**Argument Types:** Real
String
Symbol
**Result Type(s):** Real
String
Symbol
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | | | |
|---|---|---|---|---|---|
| $Real_1$ | $Symbol_2$ | $Symbol_3$ | → | $Real_4$ | $Symbol_3$ |
| $Real_1$ | $Symbol_2$ | $String_3$ | → | $Real_4$ | $String_3$ |
| $Real_1$ | $String_2$ | $Symbol_3$ | → | $Real_4$ | $Symbol_3$ |
| $Real_1$ | $String_2$ | $String_3$ | → | $Real_4$ | $String_3$ |

This operation performs a unit conversion of the value in $Real_1$ from the unit described by $Symbol_2$ or $String_2$ to the unit described by $Symbol_3$ or $String_3$. Refer to Built-In Units for information about the different units supported by default by Halcyon Calc. If you specify the unit as a string, you can use combinations of units to create more complex units like "ft / s ^ 2". The rules for these units are:

1. You can use a divide operator zero or one times in the unit expression.
2. You can use the multiply operator as many times as you would like in the unit expression. If the unit expression has a divide operator, you can use the multiple operator as many times as you would like in the numerator and the denominator of the unit expression.
3. Any individual unit in the unit expression can be exponentiated by an integer value which is 1 or greater.
4. You can use the constant 1 as the numerator in a unit expression as long as it is immediately followed by the divide operator. This lets you create units like "1 / s".

Before performing any requested conversion, the units are checked for compatibility. All units, including the built-in units, are a combination of eight base quantities:

| Base Quantity | Base Units |
|---|---|
| Length | meter (m) |
| Mass | kilogram (kg) |
| Time | second (s) |
| Electric Current | ampere (A) |
| Thermodynamic Temperature | Kelvin (°K) |
| Luminous Intensity | candela (cd) |

| | |
|---|---|
| Amount of Substance | mole (mol) |
| User Defined | ? |

The user defined quantity can be used to represent any base quantity you would like which is not easily expressible in terms of the other seven. This is particularly useful with user defined units. You can define your own unit by creating a symbol with the name of your unit. The value of the symbol must be a list with two items. The first item must be a real value which is the conversion factor. The second item must be a string or symbol which represents the units for that conversion factor. For example, to create a unit for a decade, you can store the list { 10 "yr" } into the symbol 'dec'. The list says that a decade is 10 years. Once you have stored this value, you can then do conversions from hours to decades or even feet per second to astronomical units per decade.

Conversions of temperatures are a bit more complicated because the zero point varies. If converting from Celsius to Fahrenheit, the conversion will take into account the fact that these two scales have a different zero value on their respective scales. But, if you convert degrees Celsius per second to degrees Fahrenheit per second, the different zero value is ignored and only the different value of a single degree is considered.

Also, there are units built in like arc seconds and degrees which measure angles. These units are inherently dimensionless. That means you can convert meters per degree to feet if you like. Because degree is dimensionless, the calculator considers these units to be compatible but this would be a pretty meaningless conversion. Be careful when using angles because the calculator cannot easily validate that your conversion makes sense.

With built-in units, you can prefix any of these SI prefixes:

| Prefix | Name | Factor |
|---|---|---|
| E | exa | $10^{18}$ |
| P | peta | $10^{15}$ |
| T | tera | $10^{12}$ |
| G | giga | $10^{9}$ |
| M | mega | $10^{6}$ |
| k or K | kilo | $10^{3}$ |
| h or H | hecto | $10^{2}$ |
| D | deka | $10^{1}$ |
| d | deci | $10^{-1}$ |
| c | deci | $10^{-2}$ |
| m | deci | $10^{-3}$ |
| µ | micro | $10^{-6}$ |
| n | nano | $10^{-9}$ |

| | | |
|---|---|---|
| p | pico | $10^{-12}$ |
| f | femto | $10^{-15}$ |
| a | atto | $10^{-18}$ |

Note that if putting this prefix on a built-in unit causes it to then match a different built-in unit, then the calculator matches the built-in unit without the prefix. Also, you can create units like "Mft" for mega-feet and the calculator will treat this as millions of feet even though the prefix doesn't make much sense on non-SI units. Finally, the calculator will not automatically apply these prefixes to your own custom units. You can always define more custom units with the prefix so if you want to convert to "kilo-decades", you can create custom unit called 'kdec'.

# CORR

**Member Of Menu:** Stat
**Argument Types:** None
**Result Type(s):** Real
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

→ $Real_1$

This operation calculates the correlation between the dependent and independent columns in the statistics data. It expects to find a variable called ∑DAT which has a set of statistics values stored in a real matrix. Each row in the real matrix represents a single set of samples. Each column contains one of the set of values associated with a single sample. It also expects to find a variable called ∑PAR which should be a list of four real values. The first two real values in the list is the column number of the independent and dependent columns. If the ∑PAR variable does not exist, then the operation assumes it should use column 1 and column 2 from the statistics data.

# COS

**Member Of Menu:** Trig
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the cosine function of its argument. Note that the interpretation of the input argument depends on the whether the DEG (degree) or RAD (radians) mode is on. For real valued arguments, the argument is treated as an angle in degrees if DEG is on. For real valued arguments, the argument is treated as an angle in radians if RAD is on. For complex arguments, the angle is always expected to be in radians.

# COSH

**Member Of Menu:** Logs
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the hyperbolic cosine function of its input argument. It takes real or complex input values.

# COV

**Member Of Menu:** Stat
**Argument Types:** None
**Result Type(s):** Real
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

→ $Real_1$

This operation calculates the covariance between the dependent and independent columns in the statistics data. It expects to find a variable called ∑DAT which has a set of statistics values stored in a real matrix. Each row in the real matrix represents a single set of samples. Each column contains one of the set of values associated with a single sample. It also expects to find a variable called ∑PAR which should be a list of four real values. The first two real values in the list is the column number of the independent and dependent columns. If the ∑PAR variable does not exist, then the operation assumes it should use column 1 and column 2 from the statistics data.

# CRDIR

**Member Of Menu:** Memory
**Argument Types:** Symbol
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Symbol_1$ | → |

This operation takes a symbol and creates a directory with that name in the current directory. Directories are good ways to group related symbols together. Most times a symbol is looked up, the current directory is searched and then all parents are searched until the HOME directory is reached. So, you can rely on this behaviour to allow subdirectories to inherit some values from their parents.

In order to change to directory, you can select the "User" button on the calculator and press the button associated with the directory. Or you can just type the name of the directory and press Enter.

Note that directories are only supported on Halcyon Calc. Halcyon Calc Lite does not support directories.

# CROSS

**Member Of Menu:** Array
**Argument Types:** Array
**Result Type(s):** Array
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

$Array_1$ $Array_2$ → $Array_3$

This operation calculates the cross product of the input arrays. The input arrays must be vectors with three values and the result will be a vector with three values.

# C→R

**Member Of Menu:** Trig
Complex
Array
**Argument Types:** Complex
Array
**Result Type(s):** Real
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $(Real_1, Real_2)$ | → | $Real_1$ | $Real_2$ |
| $Array_1$ | → | $Array_2$ | $Array_3$ |

This function takes a complex value and returns the individual components of that complex value. It pushes the real component to the stack first followed by the imaginary component.

The operation also works with matrices and vectors. The result is two matrices or vectors with same dimension as the input argument. $Array_2$ is the real components of each value from the input array and $Array_3$ is the imaginary component of the input array. If the input array is real valued, then $Array_3$ is populated with zeroes.

# DEC

**Member Of Menu:** Binary
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation puts the calculator in decimal mode. Integer values are displayed in base 10. The integer 100 decimal will be shown as "# 100d". The "#" at the start indicates that the number is an integer and the trailing "d" indicates the number is displayed in decimal mode.

When entering integers, the final character normally indicates what base to use when reading that value. The characters are d for decimal, h for hexadecimal, o for octal and b for binary. Omitting that character in decimal mode will result in the calculator assuming the value should be interpreted as a decimal value.

# DEG

**Member Of Menu:** Mode
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation puts the calculator in degree mode. What this means is that any other operations which operates on angles (like trig functions), the angle is assumed to be expressed in degrees. Also, operations which return an angle will return an angle expressed in degrees. Note that this is only true of real valued input and output values. If the input or output value is complex, it is always assumed to be expressed in radians.

# DEPTH

**Member Of Menu:** Stack
**Argument Types:** None
**Result Type(s):** Real
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

→ $Real_1$

This operation pushes a real number onto the stack which is the number of items on the stack (not including this new real number now on the stack). If the stack was empty when this operation was executed, a "0" will be pushed onto the stack.

# DET

**Member Of Menu:** Array
**Argument Types:** Array
**Result Type(s):** Real
Complex
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Array_1$ | → | $Real_2$ |
| $Array_1$ | → | $Complex_2$ |

This operation calculates the determinant of its input argument. The argument must be a square matrix. An error will be returned if the input is a vector or a non-square matrix.

The result will be complex if the matrix is complex. Otherwise, the result will be real.

# DO

**Member Of Menu:** Branch
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

The DO operation is used to define a loop structure within a program. It is combined with the UNTIL and END operation to define the boundaries of the loop.

The normal way DO is used is:

« ... DO ... operations ... UNTIL ... operations ... END ... »

The operations between DO and UNTIL are the loop operations and the operations between UNTIL and END are the test operations which determine whether to loop back to the beginning of the loop operations. When END is reached, the top of the stack is popped. A real value is expected. If the real value is 0 (false), then execution loops back to the operation following DO. If the real value is non-zero (true), then execution continues after END and the loop terminates.

With a DO loop, the loop operations are evaluated at least once and will continue to execute until the test operations return a true value.

An error will be raised if this operation is used outside of program execution context.

# DOT

**Member Of Menu:** Array
**Argument Types:** Array
**Result Type(s):** Real
Complex
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Array_1$ | $Array_2$ | → | $Real_3$ |
| $Array_1$ | $Array_2$ | → | $Complex_3$ |

This operation calculates the dot product of its arguments. The inputs must be vectors and they must have the same number of values.

The result is the sum of the product of each pair of values from the input vectors. If one or more of the input vectors is complex, the result will be complex. Otherwise the result will be real.

# DRAW

**Member Of Menu:** Plot
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation will plot the current equation which was specified with the STEQ operation and according to the current values from the PPAR symbol which provides the plot parameters. The plot parameters determines the portion of the X/Y coordinates to plot, the independent variable (X axis) in the equation, the resolution of the plot and finally, the position of the X and Y axis in the plot. Each of these can be adjusted with one or more operations in the Plot menu or through direct manipulation of the PPAR value itself. If PPAR does not exist, then a default PPAR will be stored and used in the plot. The defaults are reasonable for all values except the independent variable which you will want to set using the INDEP operation.

When DRAW is executed, the stack will disappear from the display and be replaced with a plot view. The axes may be visible depending on where they are drawn and the portion of the X/Y coordinate space you are viewing. A plus cursor will be drawn in the center of the display. The calculator will begin evaluating the equation for all points visible on the X axis and plotting those points on the display. When you want to return to normal stack view, press the "Attn" button.

While in plot view, you can use the arrow keys just below the plot view to move the cursor around. The current position of the cursor in X/Y coordinates will be displayed for a few seconds after the cursor's position changes. This is helpful for understanding where you are in the coordinate space. You can press the INS button in this mode. This will push the current cursor's position onto the stack as a complex number where X is the real value and Y is the imaginary value. This is a great way to get approximations for the solver when looking for maxima, minima or roots of an equation.

You can also interact with the plot directly. Use pinch gestures to zoom in and out. Slide the plot in any direction to pan through the coordinate space. Double tap to quickly move the cursor to a point. Drag the cursor directly to move it around the plot view. Double tap on the cursor to push its current location onto the stack as a complex number. Finally, rotate your device to landscape mode and the plot view will be full screen. All buttons will be inaccessible but you will have much higher resolution for your plot and all of the direct interactions with the plot will be available. Note that zoom, pan and rotate operations cause the plot to be recalculated.

# DROP

**Calculator Key:** Drop

**Member Of Menu:** None
**Argument Types:** Any
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Item_1$ | → |

This operation removes the item from the top of the stack.

# DROP2

**Member Of Menu:** Stack
**Argument Types:** Any
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**


This operation removes the top two items from the stack.

# DROPN

**Member Of Menu:** Stack
**Argument Types:** Any
Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Item_1$ ... $Item_n$ $Real_{n+1}$ →

This operation takes a real number from the top of the stack which indicates how many other items should be popped from the stack. When completed, the real number and n other items (based on the value of that real number) will be removed from the stack.

# DUP

**Calculator Key:** Enter

**Member Of Menu:** Stack
**Argument Types:** Any
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Item_1$ | → | $Item_1$ | $Item_1$ |

This operation creates a copy of the item at the top of the stack and pushes that copy onto the stack.

Note that the "Enter" key on the calculator executes the DUP operation if there is no entry in progress. If there is an entry in progress, then DUP is not executed and instead the entry is parsed.

# DUP2

**Member Of Menu:** Stack
**Argument Types:** Any
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Item_1$ $Item_2$ → $Item_1$ $Item_2$ $Item_1$ $Item_2$

This operation makes copies of the top two items on the stack and pushes both copies onto the stack.

# DUPN

**Member Of Menu:** Stack
**Argument Types:** Any
Real
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Item_1$ ... $Item_n$ $Real_{n+1}$ → $Item_1$ ... $Item_n$ $Item_1$ ... $Item_n$

This operation takes a real number from the top of the stack which indicates how many items from the stack to duplicate. Copies of each of those items will be made an pushed onto the stack.

# D→R

**Member Of Menu:** Trig
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function takes a real number which is an angle expressed in degrees and converts it to an angle expressed in radians.

# ELSE

**Member Of Menu:** Branch
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation is used in conjunction with the IF or IFERR operations. See those pages for more details.

# END

**Member Of Menu:** Branch
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation is used in conjunction with the IF, IFERR, DO and WHILE operations. See those pages for more details.

# ENG

**Member Of Menu:** Mode
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Real_1$ | → |

This operation enables the engineering format for real numbers on the calculator. The engineering format takes a real argument which is the number of digits after the left most digit to display. When enabled, there will always be that many digits shown after the left most digit, even if they are all 0's. Also, exponential format is always used in engineering format. Also, the exponent in engineering format is always evenly divisible by 3.

# ERRM

**Member Of Menu:** Control
**Argument Types:** None
**Result Type(s):** String
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

→ $String_1$

This operation pushes a string onto the stack which is the error message most recently raised by the calculator. The set of possible strings is described on the ERRN operation page.

# ERRN

**Member Of Menu:** Control
**Argument Types:** None
**Result Type(s):** Integer
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

→ $\text{Integer}_1$

The ERRN operation returns a numerical value which represents the most recently raised error by the calculator. When used with the IFERR operation within a program, you can write error handlers for different error conditions.

The error numbers currently defined are:

| Value | Message |
|---|---|
| 0 | No error (only seen if you have *never* raised an error on the calculator) |
| 1 | Insufficient Memory |
| 3 | Undefined Local Name |
| 257 | No Room For UNDO |
| 258 | Can't Edit CHR(0) |
| 259 | Improper User Function |
| 260 | No Current Equation |
| 261 | No Room To ENTER |
| 262 | Syntax Error |
| 286 | Invalid PPAR |
| 287 | Non-Real Result |
| 288 | Unable To Isolate |
| 289 | HALT Not Allowed |
| 292 | UNDO Disabled |
| 293 | Command Stack Disabled |
| 296 | Wrong Argument Count |
| 297 | Circular Reference |
| 298 | Directory Not Allowed |
| 299 | Non-Empty Directory |
| 513 | Too Few Arguments |

| | |
|---|---|
| 514 | Bad Argument Type |
| 515 | Bad Argument Value |
| 516 | Undefined Name |
| 517 | LAST Disabled |
| 769 | Positive Underflow |
| 770 | Negative Underflow |
| 771 | Overflow |
| 772 | Undefined Result |
| 773 | Infinite Result |
| 1281 | Invalid Dimension |
| 1537 | Invalid ∑DAT |
| 1538 | Nonexistent ∑DAT |
| 1539 | Insufficient ∑ Data |
| 1540 | Invalid ∑PAR |
| 2561 | Bad Guess(es) |
| 2562 | Constant? |
| 2817 | Invalid Unit String |
| 2818 | Inconsistent Units |

Some of these errors are here for future use and may not be generated by the current version of the calculator

# EVAL

**Calculator Key:** Eval

**Member Of Menu:** None
**Argument Types:** Any
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Item_1$ | → | $Item_2$ |

This operation takes an item from the stack and evaluates it. For most types of stack items, evaluation does not change anything. For example, if 8 was on the stack before EVAL was executed, the result of EVAL is also 8.

However, for symbols, expressions and programs, the result may be different from the input. If a symbol is on the stack, then this operation will lookup that symbol, traversing directories from the current directory to the HOME directory. If it finds that symbol, it will replace it with the value of that symbol.

If an expression is on the stack, then any symbols in the expression will be evaluated as described above. The value of those symbols will be substituted in the expression. Any operations in the expression will be executed as long as the arguments for those expressions is known (for example a real value and not just a symbol). The final result may still be an expression or may be some other type, like a real, depending on the evaluation of the expression.

If a program is on the stack, then the program is executed. This may actually result in more values being popped off the stack or multiple values being pushed onto the stack. So, the stack diagram above may not be accurate depending on the program.

Note that EVAL will not substitute the values of the global constants e, i and π. If you require that functionality, use the →NUM operation.

# EXGET

**Member Of Menu:** Algebra
**Argument Types:** Real
Expression
**Result Type(s):** Real
Complex
Symbol
Expression
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$\text{Expression}_1$ $\text{Real}_2$ → $\text{Item}_3$

The EXSUB operation is used to retrieve a portion of an expression. The real argument identifies the subexpression in the expression to retrieve. An expression is a sequence of real numbers, complex numbers, symbols and operations. If you read your expression from left to right, the left most item in that sequence is 1 and each item after that is the next number in the sequence. If the item at this offset is a real number, complex number or symbol, then the subexpression is that item itself. If the item at this offset is an operation, then the subexpression is the operation and its arguments.

So, for the expression 'SIN(X+10)', the subexpressions are:

| Index | Subexpression |
|---|---|
| 1 | 'SIN(X+10)' Expression |
| 2 | Symbol X |
| 3 | 'X+10' Expression |
| 4 | Real Number 10 |

The result of EXSUB is a real number, complex number, symbol or expression depending on what the subexpression is at that index. In the example able, EXGET would return 'X+10' if you look for the third subexpression.

# EXP

**Member Of Menu:** Logs
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the reverse base e logarithm of its input argument. Assuming that the input argument is x, this is equivalent to ex. Real and complex numbers are valid inputs to this operation.

# EXPAN

**Member Of Menu:** Algebra
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Symbol
Expression
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Symbol_2$ |
| $Expression_1$ | → | $Expression_2$ |

The EXPAN operation expands an expression using a series of different strategies:

- If the expression has a multiplication or division and one of the arguments to that is a addition or subtraction operation, it will distribute that multiplication or division. So, '(X+Y)*Z' will become 'X*Z+Y*Z'. Similarly, '(X-Y)/Z' will become 'X/Z-Y/Z'.
- If the expression has a power operation and the exponent of that operation is an addition or subtraction operation, it will turn that into a pair of power operation factors. So, 'X^(Y+Z)' will become 'X^Y*X^Z'. Similarly, 'X^(Y-Z)' will become 'X^Y/X^Z'.
- If the expression has a power operation and the exponent of that operation is a positive integer, it will pull one factor out of that power operation. So, 'X^10' will become 'X*X^9'.
- If the expression is a square of addition or subtraction, the square will be expanded. So, 'SQ(X+Y)' or '(X+Y)^2' will become 'X^2+2*X*Y+Y^2'. Similarly, 'SQ(X-Y)' or '(X-Y)^2' will become 'X^2-2*X*Y+Y^2'.

Note that the EXPAN operation only perform a single expansion on the expression. The first rule above which it can apply, it will and then it returns this slightly more expanded expression. This behaviour differs from COLCT which fully collects an expression as much as possible on a single execution.

# EXPM

**Member Of Menu:** Logs
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the equivalent of (EXP(x) - 1) where x is the the argument from the stack. It may appear to be redundant since the EXP operation seems to be capable of everything this operation can do, and more but this operation is much more accurate for values near 0. So, in some cases, this operation is preferred.

However, this operation is more limited since it does not handle complex numbers.

# EXSUB

**Member Of Menu:** Algebra
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Expression
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | | |
|---|---|---|---|---|
| $Expression_1$ | $Real_2$ | $Real_3$ | → | $Expression_4$ |
| $Expression_1$ | $Real_2$ | $Complex_3$ | → | $Expression_4$ |
| $Expression_1$ | $Real_2$ | $Symbol_3$ | → | $Expression_4$ |
| $Expression_1$ | $Real_2$ | $Expression_3$ | → | $Expression_4$ |

The EXSUB operation is used to substitute a portion of an expression with the item from the top of the stack. The real argument identifies the subexpression in the expression to replace. An expression is a sequence of real numbers, complex numbers, symbols and operations. If you read your expression from left to right, the left most item in that sequence is 1 and each item after that is the next number in the sequence. If the item at this offset is a real number, complex number or symbol, then the subexpression is that item itself. If the item at this offset is an operation, then the subexpression is the operation and its arguments.

So, for the expression 'SIN(X+10)', the subexpressions are:

| Index | Subexpression |
|---|---|
| 1 | 'SIN(X+10)' Expression |
| 2 | Symbol X |
| 3 | 'X+10' Expression |
| 4 | Real Number 10 |

To use EXSUB on this expression, you would specify a number from 1 to 4 depending on which subexpression you wanted to substitute.

In the above expression, if you wanted to change the argument to the SIN operation to 'COS(X)', you would specify 3 as the position put 'COS(X)' at the top of the stack. The EXSUB command would return 'SIN(COS(X))'

# FACT

**Member Of Menu:** Real
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function returns the factorial of its input parameter. For integer input values greater than or equal to 0, that means that x! is returned where x is the input value. However, FACT returns a value for all real values between -1 and 170, including non-integer values. It does so using the Gamma function. Assuming that the input value is x, the result then is Γ(x+1). For all other values, an infinite result error is returned.

# FC?

**Member Of Menu:** Test
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes a real value between 1 and 64 and returns 1 if the associated bit in the calculator flags is 0. Otherwise, the operation returns 0. The flags are unchanged by this operation.

# FC?C

**Member Of Menu:** Test
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes a real value between 1 and 64 and returns 1 if the associated bit in the calculator flags is 0. Otherwise, the operation returns 0. Also, the bit in the calculator flags is set to 0 after testing its value.

# FIX

**Member Of Menu:** Mode
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Real_1$ | → |

This operation enables the fixed format for real numbers on the calculator. The fixed format takes a real argument which is the number of digits to display to the right of the decimal point. When enabled, there will always be that many digits shown to the right of the decimal place, even if they are all 0's. If more than 12 digits are required to represent the number, then the number will be shown in exponential format. Similarly, if the number is too small to be represented at all in the number of decimal digits specified, it will be shown in exponential format.

# FLOOR

**Member Of Menu:** Real
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

Given a real valued input, this function returns the largest integer which is less than or equal to the input value.

# FOR

**Member Of Menu:** Branch
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | $Real_2$ | → |

The FOR operation is used to define a loop structure within a program. It is combined with the NEXT or STEP operation to define the boundaries of the loop.

The normal way FOR is used is one of the following:

```
« ... Real1 Real2 FOR Symbol3 ... operations ... NEXT ... »
```

or

```
« ... Real1 Real2 FOR Symbol3 ... operations ... Real4 STEP ... »
```

When FOR is executed, it pops two values of the stack which should be real values. $Real_1$ is the starting value for the loop counter and $Real_2$ is the ending value for the loop counter. Immediately following the FOR operation is a symbol name. A local variable which is valid only in the body of the loop is then created with that name which contains the loop counter. This allows you to access the loop counter in the body of the loop.

When NEXT is executed, the loop counter is incremented by one. If the loop counter is less than or equal to the ending value, then execution jumps back to the operation following FOR and executes the body of the loop again. If the value is greater than the ending value after incrementing the loop counter, then execution continues after the NEXT operation, exiting the loop.

The STEP operation can be used in place of NEXT to specify an increment for the loop counter which is not one. When STEP is executed the top of the stack is popped. It expects to find a real value there. It adds that real value to the loop counter and then tests to see if the loop counter has reached the end. For a negative increment, the loop stops when the counter is less than the ending value. For a positive increment, the loop stops when the counter is greater than the ending value. If not yet at the end, it loops back to the operation following FOR. Otherwise, it exits the loop and execution continues after the STEP operation.

An error will be raised if this operation is used outside of program execution context.

# FORM

**Member Of Menu:** Algebra
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Expression
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | | |
|---|---|---|---|---|
| $Real_1$ | → | $Expression_2$ | | |
| $Real_1$ | → | $Expression_2$ | $Real_3$ | $Expression_4$ |
| $Complex_1$ | → | $Expression_2$ | | |
| $Complex_1$ | → | $Expression_2$ | $Real_3$ | $Expression_4$ |
| $Symbol_1$ | → | $Expression_2$ | | |
| $Symbol_1$ | → | $Expression_2$ | $Real_3$ | $Expression_4$ |
| $Expression_1$ | → | $Expression_2$ | | |
| $Expression_1$ | → | $Expression_2$ | $Real_3$ | $Expression_4$ |

This operation takes an expression and allows you to interactively modify that expression in a way that does not change the way the expression would evaluate. For example, adding one and subtracting one to an expression will not change the value of that expression for any given inputs. This is one of the many ways you can modify an expression.

Under most circumstances, the input is a complex expression of some kind. However, you can pass it a real number, complex number or symbol. You can perform some operations on even simple "expressions" like that.

The output in general is a modified version of the input expression. Again, it is possible to simply the expression down to a real number, complex number or a symbol but most times the output will continue to be an expression.

When you execute this operation, the display clears and the input expression is shown in the middle of the display. The left most component of the expression will be highlighted. This indicates what component of the expression is selected and you can operate on. If the selected component is an operation, then you can operate on the sub-expression which is this operation and its arguments. If the selected component is a real number, complex number or symbol, then you can operate on that component independently.

With this operation active, the "FORM" menu buttons are displayed. Among these buttons are the [→] and [←] buttons which allow you to move the selected component of the expression to the left or the right. If you have a hardware keyboard attached to your device, you can also use the left and right cursor keys to move the selected component.

Once you have selected the component you want to operate on, then you can press one or more menu buttons to make a change to that selected component. When you are done, you can press the "Attn" button to leave the interactive editing of the expression. Your edited expression will be pushed onto the stack.

Below is a description of all of the different kinds of sub-operations you can perform in interactive expression editing mode.

# Common Sub-Operations:

The following sub-operations are available regardless of what you have selected in the current expression.

| Sub-Operation | Description |
|---|---|
| COLCT | This performs the equivalent of the COLCT operation on the sub-expression selected. Normally COLCT performs the action on a complete expression but when executed this way, you can perform it on a subset of the complete expression. |
| EXPAN | This performs the equivalent of the EXPAN operation on the sub-expression selected. Normally EXPAN performs the action on a complete expression but when executed this way, you can perform it on a subset of the complete expression. |
| LEVEL | This displays a popup telling you the level of the currently selected component of the expression. The outer-most operation is always level 1 in an expression. The inputs to at operation are at level 2. If those inputs are themselves sub-expressions, the inputs to that sub-expression is level 3, etc. |
| EXGET | This performs the equivalent of the EXGET operation on the sub-expression selected. When selected, you will leave interactive expression editing mode and go back to normal calculator mode. The expression is pushed to the stack followed by the index of the sub-expression you had selected and finally the sub-expression itself. Normally when you leave interactive editing mode, only the edited expression is pushed to the stack but if you exit this way, you will have these three values pushed onto the stack. |
| [←] | Move the selected component of the expression to the left. If the left-most component is already selected, nothing will happen. |
| [→] | Move the selected component of the expression to the right. If the right-most component is already selected, nothing will happen. |

# Conditional Sub-Operations:

The following sub-operations depend on what the input expression looks like. A sample input expression will be shown before and after the action. The expression before the action will have a component highlighted which is what you need to select in order to take that action. If you do not see an action you want to perform, you have the wrong component selected. If you do see an action as an option, that does not necessarily mean that it can actually perform the edit. If you select it and nothing happens, it could not perform that action.

←→ Commute the inputs of a sub-expression

| Before Edit | After Edit |
|---|---|
| X + Y | Y + X |
| -X + Y | Y - X |
| X - Y | -Y + X |
| -X - Y | -Y - X |
| X * Y | Y * X |
| INV(X) * Y | Y / X |
| X / Y | INV(Y) * X |
| INV(X) / Y | INV(Y) * INV(X) |

←A Associate to the left

| Before Edit | After Edit |
|---|---|
| X + (Y + Z) | (X + Y) + Z |
| X + (Y - Z) | (X + Y) - Z |
| X - (Y + Z) | (X - Y) - Z |
| X - (Y - Z) | (X - Y) + Z |
| X * (Y * Z) | (X * Y) * Z |
| X * (Y / Z) | (X * Y) / Z |
| X / (Y * Z) | (X / Y) / Z |
| X / (Y / Z) | (X / Y) * Z |
| X ^ (Y * Z) | (X ^ Y) ^ Z |

A→ Associate to the right

| Before Edit | After Edit |
|---|---|
| (X + Y) + Z | X + (Y + X) |
| (X - Y) + Z | X - (Y - Z) |
| (X + Y) - Z | X + (Y - Z) |
| (X - Y) - Z | X - (Y + Z) |

| | |
|---|---|
| (X * Y) * Z | X * (Y * Z) |
| (X / Y) * Z | X / (Y / Z) |
| (X * Y) / Z | X * (Y / Z) |
| (X / Y) / Z | X / (Y * Z) |
| (X ^ Y) ^ Z | X ^ (Y * Z) |

Distribute prefix

**→()**

| Before Edit | After Edit |
|---|---|
| -(X + Y) | -X - Y |
| -(X - Y) | -X + Y |
| -(X * Y) | -X * Y |
| -(X / Y) | -X / Y |
| -LOG(X) | LOG(INV(X)) |
| -LN(X) | LN(INV(X)) |
| INV(X * Y) | INV(X) / Y |
| INV(X / Y) | INV(X) * Y |
| INV(X ^ Y) | X ^ -Y |
| INV(ALOG(X)) | ALOG(-X) |
| INV(EXP(X)) | EXP(-X) |

Distribute to the left

**←D**

| Before Edit | After Edit |
|---|---|
| (X + Y) * Z | X * Z + Y * Z |
| (X - Y) * Z | X * Z - Y * Z |
| (X + Y) / Z | X / Z + Y / Z |
| (X - Y) / Z | X / Z - Y / Z |
| (X * Y) ^ Z | X ^ Z * Y ^ Z |
| (X / Y) ^ Z | X ^ Z / Y ^ Z |

Distribute to the right

| Before Edit | After Edit |
|---|---|
| X * (Y + Z) | X * Y + X * Z |
| X * (Y - Z) | X * Y - X * Z |
| X / (Y + Z) | INV((INV(X) * Y) + INV(X) * Z)) |
| X / (Y - Z) | INV((INV(X) * Y) - INV(X) * Z)) |
| X ^ (Y + Z) | X ^ Y * X ^ Z |

**D→**

| | |
|---|---|
| X ^ (Y - Z) | X ^ Y / X ^ Z |
| LOG(X * Y) | LOG(X) + LOG(Y) |
| LOG(X / Y) | LOG(X) - LOG(Y) |
| ALOG(X + Y) | ALOG(X) * ALOG(Y) |
| ALOG(X - Y) | ALOG(X) / ALOG(Y) |
| LN(X * Y) | LN(X) + LN(Y) |
| LN(X / Y) | LN(X) - LN(Y) |
| EXP(X + Y) | EXP(X) * EXP(Y) |
| EXP(X - Y) | EXP(X) / EXP(Y) |

**←M**

Merge left factors

| Before Edit | After Edit |
|---|---|
| (X * Y) + (X * Z) | X * (Y + Z) |
| (X * Y) - (X * Z) | X * (Y - Z) |
| (X ^ Y) * (X ^ Z) | X ^ (Y + Z) |
| (X ^ Y) / (X ^ Z) | X ^ (Y - Z) |
| LN(X) + LN(Y) | LN(X * Y) |
| LN(X) - LN(Y) | LN(X / Y) |
| LOG(X) + LOG(Y) | LOG(X * Y) |
| LOG(X) - LOG(Y) | LOG(X / Y) |
| EXP(X) * EXP(Y) | EXP(X + Y) |
| EXP(X) / EXP(Y) | EXP(X - Y) |
| ALOG(X) * ALOG(Y) | ALOG(X + Y) |
| ALOG(X) / ALOG(Y) | ALOG(X - Y) |

**M→**

Merge right factors

| Before Edit | After Edit |
|---|---|
| (X * Z) + (Y * Z) | (X + Y) * Z |
| (X / Z) + (Y / Z) | (X + Y) / Z |
| (X * Z) - (Y * Z) | (X - Y) * Z |
| (X / Z) - (Y / Z) | (X - Y) / Z |
| (X ^ Z) * (Y ^ Z) | (X * Y) ^ Z |
| (X ^ Z) / (Y ^ Z) | (X / Y) ^ Z |

**DNEG**

Negate twice

| Before Edit | After Edit |
|---|---|

| | | |
|---|---|---|
| | X | -(-X) |

**-()**

Double negate and distribute

| Before Edit | After Edit |
|---|---|
| X + Y | -(-X - Y) |
| X - Y | -(-X + Y) |
| -X + Y | -(X - Y) |
| -X - Y | -(X + Y) |
| X * Y | -(-X * Y) |
| -X * Y | -(X * Y) |
| X / Y | -(-X / Y) |
| -X / Y | -(X / Y) |
| LOG(X) | -(LOG(INV(X))) |
| LOG(INV(X)) | -(LOG(X)) |
| LN(X) | -(LN(INV(X))) |
| LN(INV(X)) | -(LN(X)) |

**DINV**

Invert twice

| Before Edit | After Edit |
|---|---|
| X | INV(INV(X)) |

**1/()**

Double invert and distribute

| Before Edit | After Edit |
|---|---|
| X * Y | INV(INV(X) / Y) |
| X / Y | INV(INV(X) * Y) |
| X ^ Y | INV(X ^ -Y) |
| X ^ -Y | INV(X ^ Y) |
| ALOG(X) | INV(ALOG(-X)) |
| ALOG(-X) | INV(ALOG(X)) |
| EXP(X) | INV(EXP(-X)) |
| EXP(-X) | INV(EXP(X)) |

***1**

Multiply by one

| Before Edit | After Edit |
|---|---|
| X | X * 1 |

**/1**

Divide by one

| Before Edit | After Edit |
|---|---|
| X | X / 1 |

**^1**

To the power of one

| Before Edit | After Edit |
|---|---|
| X | X ^ 1 |

**+1-1**

Plus one minus 1

| Before Edit | After Edit |
|---|---|
| X | X + 1 - 1 |

**L***

Replace log of a power with a product of logs

| Before Edit | After Edit |
|---|---|
| LOG(X ^ Y) | LOG(X) * Y |
| LN(X ^ Y) | LN(X) * Y |

**L()**

Replace product of logs with a log of power

| Before Edit | After Edit |
|---|---|
| LOG(X) * Y | LOG(X ^ Y) |
| LN(X) * Y | LN(X ^ Y) |

**E^**

Replace a power product with a power of power

| Before Edit | After Edit |
|---|---|
| ALOG(X * Y) | ALOG(X) ^ Y |
| ALOG(X / Y) | ALOG(X) ^ INV(Y) |
| EXP(X * Y) | EXP(X) ^ Y |
| EXP(X / Y) | EXP(X) ^ INV(Y) |

**E()**

Replace power of power with power product

| Before Edit | After Edit |
|---|---|
| ALOG(X) ^ Y | ALOG(X * Y) |
| ALOG(X) ^ INV(Y) | ALOG(X / Y) |
| EXP(X) ^ Y | EXP(X * Y) |
| EXP(X) ^ INV(Y) | EXP(X / Y) |

**AF**

Add fractions over a common denominator

| Before Edit | After Edit |
|---|---|
| A + (B / C) | (A * C + B) / C |
| (A / B) + C | (A + B * C) / B |
| (A / B) + (C / D) | (A * D + B * C) / (B * D) |
| A - (B / C) | (A * C - B) / C |
| (A / B) - C | (A - B * C) / B |
| (A / B) - (C / D) | (A * D - B * C) / (B * D) |

# FP

**Member Of Menu:** Real
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function takes a real value and returns a real value, returning only the fractional component of that input value.

# FS?

**Member Of Menu:** Test
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes a real value between 1 and 64 and returns 1 if the associated bit in the calculator flags is 1. Otherwise, the operation returns 0. The flags are unchanged by this operation.

# FS?C

**Member Of Menu:** Test
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes a real value between 1 and 64 and returns 1 if the associated bit in the calculator flags is 1. Otherwise, the operation returns 0. Also, the bit in the calculator flags is set to 0 after testing its value.

# GET

**Member Of Menu:** List
Array
**Argument Types:** Real
List
Symbol
Array
**Result Type(s):** Any
Real
Complex
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $List_1$ | $Real_2$ | → | $Item_3$ |
| $List_1$ | $List_2$ | → | $Item_3$ |
| $Array_1$ | $Real_2$ | → | $Item_3$ |
| $Array_1$ | $List_2$ | → | $Item_3$ |
| $Symbol_1$ | $Real_2$ | → | $Item_3$ |
| $Symbol_1$ | $List_2$ | → | $Item_3$ |

This operation is used to retrieve an item at a particular index within a list. The list to retrieve from may be found on the stack or it may be a list found in memory, referenced by a symbol name. In either case, the item at the real number index is retrieved and pushed onto the stack. If the index is a list and that list contains only one real value, then that real value is used as an index into the source to retrieve a value.

The operation also operates on vectors a matrices. If the item being operated on is a vector, either on the stack or referenced through a symbol, then the index can be a real value or a list with a single real value. The real or complex value at that location in the vector is pushed as the result. If the item being operated on is a matrix, then the index must be a list with two real values which represent the row and column of the item to retrieve. Again, the result will be a real or complex value.

The operation will produce an error if the index is outside the range of the list, vector or matrix.

# GETI

**Member Of Menu:** List
Array
**Argument Types:** Real
List
Symbol
Array
**Result Type(s):** Any
Real
Complex
List
Symbol
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | | | |
|---|---|---|---|---|---|
| $List_1$ | $Real_2$ | → | $List_1$ | $Real_3$ | $Item_4$ |
| $List_1$ | $List_2$ | → | $List_1$ | $List_3$ | $Item_4$ |
| $Array_1$ | $Real_2$ | → | $Array_1$ | $Real_3$ | $Item_4$ |
| $Array_1$ | $List_2$ | → | $Array_1$ | $List_3$ | $Item_4$ |
| $Symbol_1$ | $List_2$ | → | $Symbol_1$ | $List_3$ | $Item_4$ |
| $Symbol_1$ | $Real_2$ | → | $Symbol_1$ | $Real_3$ | $Item_4$ |

This operation is used to retrieve an item at a particular index within a list, vector or matrix. When operating on a list, the list to retrieve from may be found on the stack or it may be a list found in memory, referenced by a symbol name. In either case, the list or symbol is kept on the stack, the real number index is incremented and the item at the real number index is retrieved and pushed onto the stack. The intention is that incrementing the index on the stack allows quick iteration over the items on the stack. The retrieved item merely has to be removed from the top of the stack (stored elsewhere if necessary) and GETI can be executed again to get the next item in the list. If the index points to the end of the list before the operation, the index after the operation will have wrapped and begin again at one. Note that the index can be a real value or a list with a single real value when operating on lists.

If operating on a vector either directly on the stack or referenced by a symbol, the index can be a real value or a list with a single real value. The vector or symbol remains on the stack after the operation. The index is incremented to point to the next item in the vector, wrapping to the first item if at the end. Finally, the real or complex value at that location in the vector is pushed onto the stack.

If operating on a matrix either directly on the stack or referenced by a symbol, the index must be a list with two real values representing the row and column location. After execution, the matrix or symbol remains on the stack. The index is incremented such that the operation visits each column in a row and then incrementing the row count until wrapping back around to the item at the first row, first column. Finally, the real or complex value at the input row and column is

pushed onto the stack.

Also, this operation sets the 46th flag as described in the Working With Programs guide if the index wrapped. Otherwise, it clears the flag. Testing this flag can be useful in a program in order to loop over all items in the list, vector or matrix.

# HALT

**Member Of Menu:** Control
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation should be used only during execution of a program. By placing a HALT operation within a program, you can suspend execution of the program at that point. Then, you can examine the state of the stack and continue, single-step or abort execution of that program. HALT preserves execution state, even if you quit and relaunch the calculator so execution can be continued later.

# HEX

**Member Of Menu:** Binary
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation puts the calculator in hexadecimal mode. Integer values are displayed in base 16. The integer 100 hexadecimal will be shown as "# 100h". The "#" at the start indicates that the number is an integer and the trailing "h" indicates the number is displayed in hexadecimal mode.

When entering integers, the final character normally indicates what base to use when reading that value. The characters are d for decimal, h for hexadecimal, o for octal and b for binary. Omitting that character in hexadecimal mode will result in the calculator assuming the value should be interpreted as a hexadecimal value.

# HMS+

**Member Of Menu:** Trig
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This function takes arguments and returns results which express a time as hours, minutes, seconds and fractions of seconds. All inputs and outputs are interpreted as:

H.MMSS

The H value is the hour component. The two digits to the right of the decimal are the minutes component (MM) and the next to digits are the seconds component (SS) and further digits beyond those four to the right of the decimal are fractions of seconds.

This function takes two time values in this format and returns the sum of these two times, again in the the same H.MMSS format.

# HMS-

**Member Of Menu:** Trig
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This function takes arguments and returns results which express a time as hours, minutes, seconds and fractions of seconds. All inputs and outputs are interpreted as:

H.MMSS

The H value is the hour component. The two digits to the right of the decimal are the minutes component (MM) and the next to digits are the seconds component (SS) and further digits beyond those four to the right of the decimal are fractions of seconds.

This function takes two time values in this format and returns the difference of these two times, again in the the same H.MMSS format.

# HMS→

**Member Of Menu:** Trig
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function takes a real argument which describes a time as hours, minutes and seconds and converts that time into hours and fractions of hours (the decimal component). The input argument will be a number which looks like:

H.MMSS

The H value is the hour component. The two digits to the right of the decimal are the minutes component (MM) and the next to digits are the seconds component (SS) and further digits beyond those four to the right of the decimal are fractions of seconds.

# HOME

**Member Of Menu:** Memory
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation changes the current directory to the root directory, regardless of where in the symbol hierarchy the calculator was prior to execution.

# IDN

**Member Of Menu:** Array
**Argument Types:** Real
Symbol
Array
**Result Type(s):** None
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Array_2$ |
| $Array_1$ | → | $Array_2$ |
| $Symbol_1$ | → | |

This operation returns an identity matrix. An identity matrix has values zero in all positions except along the diagonal from the top left to the bottom right whose values are all one. An identity matrix is always a square matrix.

If the argument is a real value, then the result is an identity matrix with that many rows and that many columns. If the argument is an array, it must be a square matrix. An error will be returned if it is a vector or non-square matrix. The result will be an identity matrix with the same dimensions.

Finally, if the argument is a symbol, then the value of that symbol must be a square matrix. An identity matrix of the same dimensions will then be stored at that symbol.

# IF

**Member Of Menu:** Branch
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

The IF operation can be used within program execution context to control the flow of execution through the program. It is combined with the THEN, END and possibly the ELSE operation to form a IF control structure.

The normal way IF is used is one of the following:

« ... IF ... operations ... THEN ... operations ... END ... »

or

« ... IF ... operations ... THEN ... operations ... ELSE ... operations ... END ... »

During execution, the top value of the stack will be popped when THEN is reached. THEN expects to find a real value there. If the value is non zero, then the operations following the THEN is executed. The operations which may follow an optional ELSE in the program will be skipped if the THEN block of operations were executed. If the value popped from the stack at THEN is zero, execution continues after the ELSE if it is present. In all cases, execution will continue after the END block.

An error will be raised if this operation is used outside of program execution context.

# IFERR

**Member Of Menu:** Branch
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

The IFERR operation can be used within program execution context to control the flow of execution through the program. It is combined with the THEN, END and possibly the ELSE operation to form a IFERR control structure.

The normal way IFERR is used is one of the following:

« ... IFERR ... operations ... THEN ... operations ... END ... »

or

« ... IFERR ... operations ... THEN ... operations ... ELSE ... operations ... END ... »

During execution, the operations between IFERR and THEN are executed. If in the course of executing those operations an error is raised, then execution will then jump to the operations following THEN, skipping past any other operations between IFERR and THEN. At this point, your program can attempt to handle the error which occurred. After executing the THEN block of operations, an optional block of ELSE operations will be skipped and execution will continue after the END operation.

If no error occurs during the execution of the operations between IFERR and THEN, then execution will jump to the optional ELSE block if present or just continue after END if no ELSE is present.

Note that if an operation raises as error during normal program execution, execution will stop and not be restartable. One common use for IFERR might be to do something like:

« ... IFERR ... operations ... THEN HALT END ... »

This allows you to suspend the execution of the program when an error occurs. Note that if you continue execution, it continues from the HALT operation and does not retry the operations which were skipped between IFERR and THEN. You need to manually run those operations before continuing.

An error will be raised if this operation is used outside of program execution context.

# IFT

**Member Of Menu:** Branch
**Argument Types:** Any
Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | $Item_2$ | → |

The IFT operation is an alternative to the IF operation. In this form, IFT pops two values off the stack. The real value is tested to see if it is non-zero. If it is non-zero, then the item popped off the top of the stack is evaluated. If the item is not an expression or program, then the item is just pushed back onto the stack. If the item is an expression or program, it is executed which may result in one or more items being pushed onto the stack or any number of things happening.

This would often be used like this:

« ... PushSomeCondition « ExecuteIfTrue » IFT ... »

In this case "PushSomeCondition" is one or more operations which result in a real value being pushed onto the stack. These operations may contain logical comparisons like ==. After that, a program is pushed onto the stack which itself calls "ExecuteIfTrue" in this example. When IFT is executed, the program and real value are popped and the program is executed if the real value is non-zero.

# IFTE

**Member Of Menu:** Branch
**Argument Types:** Any
Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Item_2$ | $Item_3$ | → |

The IFTE operation is an alternative to the IF operation. In this form, IFTE pops three values off the stack. The real value is tested to see if it is non-zero. If it is non-zero, then Item2 is evaluated. If the real value is zero, then Item3 is evaluated. If the item is an expression or program, it is executed which may result in one or more items being pushed onto the stack or any number of things happening.

This would often be used like this:

« ... PushSomeCondition « ExecuteIfTrue » « ExecuteIfFalse » IFTe ... »

In this case "PushSomeCondition" is one or more operations which result in a real value being pushed onto the stack. These operations may contain logical comparisons like ==. After that, a program for the true clause is pushed onto the stack which itself calls "ExecuteIfTrue" in this example followed by a program which calls "ExecuteIfFalse" for the false clause. When IFTe is executed, the two programs and real value are popped and the appropriate program is executed based on the value of the real number.

# IM

**Member Of Menu:** Complex
Array
**Argument Types:** Real
Complex
Symbol
Expression
Array
**Result Type(s):** Real
Expression
Array
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | 0 |
| $Complex_1$ | → | $Real_2$ |
| $Array_1$ | → | $Array_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function takes a complex value as input and returns the imaginary component of that complex number. If the input value is a real value, then 0 is returned since the imaginary component of a real value is 0.

This operation also works with vectors and matrices. The result is a vector or matrix with the same dimensions as the input. The result is always a real valued vector or matrix consisting of the imaginary components of the corresponding values from the input array. If the input array is real valued, then all values in the resulting array are zero.

# INDEP

**Member Of Menu:** Plot
**Argument Types:** Symbol
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Symbol_1$ | → |

This operation takes a symbol from the top of the stack. The symbol specifies the "independent variable" to use when plotting the equation which was specified with STEQ. The independent variable is calculated for each point along the X axis. The value of the equation is plotted on the Y axis.

The independent variable is stored in the plot parameters, found in a variable called PPAR in the current directory. The existing independent variable which may be specified in an existing PPAR symbol will be changed and then PPAR is stored, replacing the old value. If PPAR did not exist prior to executing these operations, a new PPAR symbol is created. Its contents will be defaults except for the independent variable which is specified from the stack.

# INV

**Calculator Key:** ■1/x

**Member Of Menu:** None
**Argument Types:** Real
Complex
Symbol
Expression
Array
**Result Type(s):** Real
Complex
Expression
Array
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Array_1$ | → | $Array_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes a real of complex value and produces its inverse. The inverse is one divided by the input value. Note that the calculator produces an infinite result error if zero is inverted.

This operation can calculate the inverse of a matrix. The input must be a square matrix which means it has the same number of rows as columns. The operation will fail if the input is a vector or non-square matrix. The result is the inverse matrix of the input matrix. If the matrix is not invertible, then the calculator produces an infinite result error.

# IP

**Member Of Menu:** Real
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function takes a real value and returns a real value, removing any fractional component of that input value. It returns only the integer part of the input value.

# ISOL

**Member Of Menu:** Solv
Algebra
**Argument Types:** Symbol
Expression
**Result Type(s):** Expression
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$\text{Expression}_1 \ \text{Symbol}_2 \ \rightarrow \ \text{Expression}_3$

This operation takes an expression and a symbol. It then searches for that symbol in that expression. The first instance of that symbol it finds in the expression is then isolated. If isolating X in the expression 'X+Y=Z', the result will be 'Z-Y'. That means the first instance of X in the expression is equal to 'Z-Y'. If the expression does not contain an "=" operation, then an implied "=0" is appended to the end of the expression.

If there is only one instance of that symbol in the expression, then the result will be an expression which contains no instances of the isolated symbol. On the other hand, if there are more than one instance of the symbol in the expression, then the result will include the symbol being isolated. For example, if 'X+Y+X=Z' is isolated for X, the result will be 'Z-X-Y'.

The expression must only contain operations which have an inverse. Each operation in this reference which has an inverse is labelled with "Invertible: Yes". Some inverse functions are periodic. In these cases, a "n1", "n2", etc symbol is inserted into the result. The "n#" symbol should be set to any integer variable (positive, negative or zero) to evaluate for a specific periodic value. In some cases, the inverse has a positive or negative result. In this case a "s1", "s2", etc symbol is inserted into the result. This variable should be set to one to evaluate the positive result or minus 1 to evaluate the negative result.

The result is also evaluated (see the EVAL operation for information about what is done during evaluation) before it is pushed onto the stack. If all symbols have real values, the result could be a real number.

# KILL

**Member Of Menu:** Control
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

The KILL operation can be used within a program or outside of program execution. When used within a program, it halts execution of the program and discards program execution state. Because execution state is discarded, the program cannot be continued or single-stepped. Also, KILL will discard any other stored execution states from previously HALT-ed programs.

When used outside of program execution context, KILL will discard all executions states from previously HALT-ed programs.

# LAST

**Member Of Menu:** Mode
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation enables or disables the ▪Last functionality on the calculator. The ▪Last function allows you to recall the values which were passed into the most recently executed operation.

# LAST

**Calculator Key:** ▪Last

**Member Of Menu:** None
**Argument Types:** None
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | |
|---|---|---|---|
| → | $Item_1$ | ... | $Item_n$ |

This operation takes no arguments from the stack. Instead, it pushes onto the stack the last arguments used in the most previously executed operation.

# LIST→

**Member Of Menu:** List
Stack
**Argument Types:** List
**Result Type(s):** Any
Real
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$\{ \text{Item}_1 \ldots \text{Item}_n \} \rightarrow \text{Item}_1 \ \ldots \ \text{Item}_n \ \text{Real}_{n+1}$

This operation expects a list at the top of the stack. It then pushes each item from that list into the stack, followed by a real number which is the number of items that were in the list.

# LN

**Member Of Menu:** Logs
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Real_1$ | → | $Complex_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the natural logarithm (base e) of its input argument. If the argument is 0, an infinite result error is produced. For positive real numbers, the result is a real but for negative real numbers, the result is complex.

# LNP1

**Member Of Menu:** Logs
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the equivalent of LN(1 + x) where x is the the argument from the stack. It may appear to be redundant since the LN operation seems to be capable of everything this operation can do, and more but this operation is much more accurate for values near 0 (ie near 1 for LN). So, in some cases, this operation is preferred.

However, this operation is more limited. It does not handle complex numbers and will fail for any input value of -1 or less.

# LOG

**Member Of Menu:** Logs
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Real_1$ | → | $Complex_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the base 10 logarithm of its input argument. If the argument is 0, an infinite result error is produced. For positive real numbers, the result is a real but for negative real numbers, the result is complex.

# LR

**Member Of Menu:** Stat
**Argument Types:** None
**Result Type(s):** Real
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| → | $Real_1$ | $Real_2$ |

This operation calculates the linear regression between the dependent and independent columns in the statistics data. It expects to find a variable called ∑DAT which has a set of statistics values stored in a real matrix. Each row in the real matrix represents a single set of samples. Each column contains one of the set of values associated with a single sample. It also expects to find a variable called ∑PAR which should be a list of four real values. The first two real values in the list is the column number of the independent and dependent columns. If the ∑PAR variable does not exist, then the operation assumes it should use column 1 and column 2 from the statistics data.

$Real_1$ is the intercept and $Real_2$ is the slope of the line calculated from the linear regression of the data. This information is also stored in the ∑PAR variable in the third and fourth values respectively in the list. These values are then used by the PREDV operation.

# MANT

**Member Of Menu:** Real
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function returns the mantissa of the input real value. Assuming that the real value is expressed in scientific notation (the actual format being used on the calculator is irrelevant but for the purposes of explanation, assume the value is in scientific notation) like so:

x.xxxxEyy

Then, the value returned will be x.xxxx, removing the exponent.

# MAX

**Member Of Menu:** Real
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This function returns the largest of the two real values it takes as input.

# MAXR

**Member Of Menu:** Real
**Argument Types:** None
**Result Type(s):** Symbol
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

→ $\text{Symbol}_1$

This operation returns a symbol which represents the largest real value which can be represented on the calculator.

# MAX∑

**Member Of Menu:** Stat
**Argument Types:** None
**Result Type(s):** Real
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| → | $Real_1$ |
| → | $Array_1$ |

This operation gets the maximum values from the statistics data. It expects to find a variable called ∑DAT which has a set of statistics values stored in a real matrix. Each row in the real matrix represents a single set of samples. Each column contains one of the set of values associated with a single sample. The operation calculates the maximum of each column of data in the real matrix and pushes the result into the stack.

If the real matrix has a single column, then a real value which is the maximum of all values in that column is pushed onto the stack. If the real matrix has two or more columns, then a real vector with the same number of columns is pushed onto the stack where each value in the vector is the maximum of values from that column.

# MEAN

**Member Of Menu:** Stat
**Argument Types:** None
**Result Type(s):** Real
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| → | $Real_1$ |
| → | $Array_1$ |

This operation calculates the mean of the statistics data. It expects to find a variable called ∑DAT which has a set of statistics values stored in a real matrix. Each row in the real matrix represents a single set of samples. Each column contains one of the set of values associated with a single sample. The operation calculates the mean of each column of data in the real matrix and pushes the result into the stack.

If the real matrix has a single column, then a real value which is the mean of all values in that column is pushed onto the stack. If the real matrix has two or more columns, then a real vector with the same number of columns is pushed onto the stack where each value in the vector is the mean of values from that column.

# MEM

**Member Of Menu:** Memory
**Argument Types:** None
**Result Type(s):** Real
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| → | $Real_1$ |

This operation pushes the amount of free memory in bytes onto the stack as a real value.

# MENU

**Member Of Menu:** Memory
**Argument Types:** Real
List
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Real_1$ | → |
| $List_1$ | → |

This operation has two different modes. If the top of the stack contains a real number, it uses that real number as an index into a table of menu buttons to open. The table of mappings can be found below:

| Real Value | Menu |
|---|---|
| 1 | Reserved |
| 2 | Binary |
| 3 | Complex |
| 4 | String |
| 5 | List |
| 6 | Real |
| 7 | Stack |
| 8 | Store |
| 9 | Memory |
| 10 | Reserved |
| 11 | Reserved |
| 12 | Reserved |
| 13 | Control |
| 14 | Branch |
| 15 | Test |
| 16 | Mode |
| 17 | Logs |
| 18 | Plot |
| 19 | Custom |
| | |

| | |
|---|---|
| 20 | Default |
| 21 | Trig |
| 22 | Solv |
| 23 | User |
| 24 | Solvr |

The reserved menus are for future expansion. Going to these menus will show a blank set of buttons.

Alternatively, this operation can be used to set the "Custom" menu buttons to a set of shortcuts which you would like quick access to. If you need quick access to SIN, LN and FACT, then you can create a list which looks like this:

{ SIN LN FACT }

After executing this operation with this list, selecting the "Custom" button on the calculator will show these three operations on these custom buttons.

You can also put any item into the list. For example, if the list contains a real number, then a button on the custom menu will have that real number on it. When that button is pressed, that real number is pushed onto the stack.

There is also a special case for a list which contains the symbol STO at the start. In this case, the list must contain symbols only. Imagine this list is passed to MENU:

{ STO X Y Z }

In this case, the custom buttons will be labelled "X=", "Y=" and "Z=". In this case, pressing one of these buttons will pop the value off of the top of the stack and store it into the named variable.

# MIN

**Member Of Menu:** Real
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This function returns the smallest of the two real values it takes as input.

# MINR

**Member Of Menu:** Real
**Argument Types:** None
**Result Type(s):** Symbol
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

→ $\text{Symbol}_1$

This operation returns a symbol which represents the smallest real value greater than zero which can be represented on the calculator.

# MIN∑

**Member Of Menu:** Stat
**Argument Types:** None
**Result Type(s):** Real
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| → | $Real_1$ |
| → | $Array_1$ |

This operation gets the minimum values from the statistics data. It expects to find a variable called ∑DAT which has a set of statistics values stored in a real matrix. Each row in the real matrix represents a single set of samples. Each column contains one of the set of values associated with a single sample. The operation calculates the minimum of each column of data in the real matrix and pushes the result into the stack.

If the real matrix has a single column, then a real value which is the minimum of all values in that column is pushed onto the stack. If the real matrix has two or more columns, then a real vector with the same number of columns is pushed onto the stack where each value in the vector is the minimum of values from that column.

# ML

**Member Of Menu:** Mode
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation enables or disables multi-line display for the item at the bottom of the stack. When disabled, the entry at the bottom of the stack will be truncated if too long to fit on a single line.

# MOD

**Member Of Menu:** Real
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This function returns the modulus or remainder of its two real valued inputs.

# NEG

**Calculator Key:** CHS

**Member Of Menu:** Complex
Real
Array

**Argument Types:** Real
Complex
Symbol
Expression
Array

**Result Type(s):** Real
Complex
Expression
Array

**Invertible:** Yes

**Valid In Expression:** Yes

**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $-Real_1$ |
| $Complex_1$ | → | $-Complex_1$ |
| $Array_1$ | → | $Array_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

Given a real or complex input, this function returns the value of that input multiplied by -1. Given a matrix or vector, this function returns a matrix or vector of the same dimension with each value multiplied by -1.

Note that the CHS (change sign) button on the calculator executes the NEG operation on the top item on the stack. However, if you press CHS while in the midst of entering something, it does not execute CHS and instead tries to negate the current entry without pushing it to the stack.

# NEXT

**Member Of Menu:** Branch
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation is used in conjunction with the START or FOR operations. See those pages for more details.

# NOT

**Member Of Menu:** Binary
Test
**Argument Types:** Real
Integer
Symbol
Expression
**Result Type(s):** Real
Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Integer_1$ | → | $Integer_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation performs a binary not operation on its integer argument and returns the integer result. If the input argument is a real value, it returns 1 if the input value is 0, otherwise it returns 0.

Note that when used in an expression on the symbol x for example, it would look like 'NOT x'.

# NUM

**Member Of Menu:** String
**Argument Types:** String
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $String_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes a string value and maps it to a real value. The first character in the string is used to lookup a real value from the table below:

| Character | Real Value |
|---|---|
| ▪ | 0 |
| <space> | 32 |
| ! | 33 |
| \ | 34 |
| # | 35 |
| $ | 36 |
| % | 37 |
| & | 38 |
| ' | 39 |
| ( | 40 |
| ) | 41 |
| * | 42 |
| + | 43 |
| , | 44 |
| - | 45 |
| . | 46 |
| / | 47 |
| 0 | 48 |

| | |
|---|---|
| 1 | 49 |
| 2 | 50 |
| 3 | 51 |
| 4 | 52 |
| 5 | 53 |
| 6 | 54 |
| 7 | 55 |
| 8 | 56 |
| 9 | 57 |
| : | 58 |
| ; | 59 |
| < | 60 |
| = | 61 |
| > | 62 |
| ? | 63 |
| @ | 64 |
| A | 65 |
| B | 66 |
| C | 67 |
| D | 68 |
| E | 69 |
| F | 70 |
| G | 71 |
| H | 72 |
| I | 73 |
| J | 74 |
| K | 75 |
| L | 76 |
| M | 77 |
| N | 78 |
| O | 79 |
| | |

| | |
|---|---|
| P | 80 |
| Q | 81 |
| R | 82 |
| S | 83 |
| T | 84 |
| U | 85 |
| V | 86 |
| W | 87 |
| X | 88 |
| Y | 89 |
| Z | 90 |
| [ | 91 |
| \ | 92 |
| ] | 93 |
| ^ | 94 |
| _ | 95 |
| ` | 96 |
| a | 97 |
| b | 98 |
| c | 99 |
| d | 100 |
| e | 101 |
| f | 102 |
| g | 103 |
| h | 104 |
| i | 105 |
| j | 106 |
| k | 107 |
| l | 108 |
| m | 109 |
| n | 110 |
| o | 111 |

| | |
|---|---|
| p | 112 |
| q | 113 |
| r | 114 |
| s | 115 |
| t | 116 |
| u | 117 |
| v | 118 |
| w | 119 |
| x | 120 |
| y | 121 |
| z | 122 |
| { | 123 |
| \| | 124 |
| } | 125 |
| ~ | 126 |
| ▦ | 127 |
| ÷ | 129 |
| × | 130 |
| √ | 131 |
| ∫ | 132 |
| ∑ | 133 |
| ▶ | 134 |
| π | 135 |
| ∂ | 136 |
| ≤ | 137 |
| ≥ | 138 |
| ≠ | 139 |
| ∝ | 140 |
| → | 141 |
| ← | 142 |
| µ | 143 |

| | |
|---|---|
| ° | 145 |
| ≪ | 146 |
| ≫ | 147 |

# N∑

**Member Of Menu:** Stat
**Argument Types:** None
**Result Type(s):** Real
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

→ $Real_1$

This operation returns the number of samples in the statistics data. It expects to find a variable called ∑DAT which has a set of statistics values stored in a real matrix. Each row in the real matrix represents a single set of samples. Each column contains one of the set of values associated with a single sample. This operation pushes the number of rows in the ∑DAT real matrix onto the stack.

# OBGET

**Member Of Menu:** Algebra
**Argument Types:** Real
Expression
**Result Type(s):** List
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$\text{Expression}_1 \quad \text{Real}_2 \quad \rightarrow \quad \text{List}_3$

The OBGET operation is used to retrieve a portion of an expression and return it as an item in a list. The real argument identifies the component in the expression to retrieve. An expression is a sequence of real numbers, complex numbers, symbols and operations. If you read your expression from left to right, the left most item in that sequence is 1 and each item after that is the next number in the sequence.

So, for the expression 'SIN(X+10)', the components are:

| Index | Component |
|---|---|
| 1 | SIN Operation |
| 2 | Symbol X |
| 3 | + Operation |
| 4 | Real Number 10 |

To use OBGET on this expression, you would specify a number from 1 to 4 depending on which component you wanted to retrieve.

If you executed OBGET on index 3 on the above expression, the list { + } would be returned since the third item is the addition operation.

# OBSUB

**Member Of Menu:** Algebra
**Argument Types:** Real
List
Expression
**Result Type(s):** Expression
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Expression_1$ $Real_2$ $List_3$ → $Expression_4$

The OBSUB operation is used to substitute a portion of an expression with the contents of a list. The real argument identifies the component in the expression to replace. An expression is a sequence of real numbers, complex numbers, symbols and operations. If you read your expression from left to right, the left most item in that sequence is 1 and each item after that is the next number in the sequence.

So, for the expression 'SIN(X+10)', the components are:

| Index | Component |
|---|---|
| 1 | SIN Operation |
| 2 | Symbol X |
| 3 | + Operation |
| 4 | Real Number 10 |

To use OBSUB on this expression, you would specify a number from 1 to 4 depending on which component you wanted to substitute.

The list argument must be a list with a single item in it. The list can contain a real number, complex number, symbol or operation. If you try to substitute a real or complex number at a position where the expression has an operation, the OBSUB command will fail. Similarly if you try to substitute an operation where there is not currently an operation.

In the above expression, if you wanted to change the addition to a subtraction, you would specify 3 as the position and the list would look like { - }. The OBSUB command would return 'SIN(X-10)'

# OCT

**Member Of Menu:** Binary
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation puts the calculator in octal mode. Integer values are displayed in base 8. The integer 100 octal will be shown as "# 100o". The "#" at the start indicates that the number is an integer and the trailing "o" indicates the number is displayed in octal mode.

When entering integers, the final character normally indicates what base to use when reading that value. The characters are d for decimal, h for hexadecimal, o for octal and b for binary. Omitting that character in octal mode will result in the calculator assuming the value should be interpreted as an octal value.

# OR

**Member Of Menu:** Binary
Test
**Argument Types:** Real
Integer
Symbol
Expression
**Result Type(s):** Real
Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Integer_1$ | $Integer_2$ | → | $Integer_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation performs a binary or operation on its two integer arguments and returns the integer result. If the input arguments are real values, it returns a 0 if both inputs are 0, otherwise it returns 1.

Note that when used in an expression on symbols x and y for example, it would look like 'x OR y'.

# ORDER

**Member Of Menu:** Memory
**Argument Types:** List
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $List_1$ | → |

This operation takes a list of symbols and re-orders those symbols in the current directory to match the order in the list. If the symbols in the current directory, shown when you press the "User" button, looks like this:

X4 X6 X2 X1 X3 X5

And the following list is passed to ORDER:

{ X3 X2 X1 X0 }

Then after executing this operation, the symbols will be in this order:

X3 X2 X1 X4 X6 X5

This demonstrates some of the details of this operation. If a symbol in the list does not exist in the current directory (like X0 in the example), it is ignored. Symbols that do exist in the current directory will match the order in the list. Following that, any symbols in the current directory which were not present in the list (X4, X5 and X6 in the example), will follow the re-ordered symbols and remain in their original order relative to each other.

# OVER

**Member Of Menu:** Stack
**Argument Types:** Any
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Item_1$ $Item_2$ → $Item_1$ $Item_2$ $Item_1$

This operation creates a copy of the item just below the top of the stack and pushes that item onto the stack.

# PATH

**Member Of Menu:** Memory
**Argument Types:** None
**Result Type(s):** List
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

→ $List_1$

This operation pushes a list onto the stack which contains the set of directories which leads to the current directory. If in the root directory, the list { HOME } is pushed onto the stack. If the current directory is DIR2 which exists inside of DIR1 below the root, then the list will be { HOME DIR1 DIR2 }.

# PERM

**Member Of Menu:** Stat
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation calculates the number of permutations given $Real_1$ items taken $Real_2$ at a time.

# PICK

**Member Of Menu:** Stack
**Argument Types:** Any
Real
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$$Item_1 \; Item_2 \; ... \; Item_{n-1} \; Item_n \; Real_{n+1} \rightarrow Item_1 \; Item_2 \; ... \; Item_{n-1} \; Item_n \; Item_1$$

This operation takes a real number which references an item on the stack. After popping that real number off of the stack, it uses that value as an index and copies that item off of the stack (1 is the top of the stack, 2 is the next item, etc). Finally, that copied item is pushed onto the top of the stack.

# PMAX

**Member Of Menu:** Plot
**Argument Types:** Complex
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Complex_1$ | → |

This operation takes a complex number from the top of the stack and interprets that complex number as X and Y coordinates. It then uses that point as the upper right hand point for a subsequent plot and stores that information in the plot parameters, found in a variable called PPAR in the current directory. The existing plot maximum which may be specified in an existing PPAR variable will be changed and then PPAR is stored, replacing the old value. If PPAR did not exist prior to executing these operations, a new PPAR variable is created. Its contents will be defaults except for the plot maximum which is specified from the stack.

# PMIN

**Member Of Menu:** Plot
**Argument Types:** Complex
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Complex_1$ | → |

This operation takes a complex number from the top of the stack and interprets that complex number as X and Y coordinates. It then uses that point as the lower left hand point for a subsequent plot and stores that information in the plot parameters, found in a variable called PPAR in the current directory. The existing plot minimum which may be specified in an existing PPAR variable will be changed and then PPAR is stored, replacing the old value. If PPAR did not exist prior to executing these operations, a new PPAR variable is created. Its contents will be defaults except for the plot minimum which is specified from the stack.

# POS

**Member Of Menu:** String
List
**Argument Types:** Any
List
String
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $String_1$ | $String_2$ | → | $Real_3$ |
| $List_1$ | $Item_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation is used to find the position of a substring within a string or the position of an item in a list. If provided two strings, it searches for the index into $string_1$ where $string_2$ can be found. The index is 1 based and POS returns 0 if the $string_2$ can not be found within $string_1$.

If a list is provided, then the item on the stack is compared to each item on the list until a match is found. If no match is found, POS returns 0. Otherwise, it returns the index within the list where the first match was found.

# PPAR

**Member Of Menu:** Plot
**Argument Types:** None
**Result Type(s):** List
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

→ $List_1$

This operation is really just a shortcut for accessing a symbol called "PPAR" from the current directory. The PPAR symbol is used to store plot parameters and its contents is a five element list of the following items:

1. The bottom left point (plot minimum) in X/Y coordinates expressed as a complex number. The default value is (-6.8, -1.5).
2. The upper right point (plot maximum) in X/Y coordinates expressed as a complex number. The default value is (6.8, 1.5).
3. The independent (X) variable to use when calculating points to plot from the equation. This item should be a symbol and its default value is "constant".
4. The resolution of the plot expressed as a real value. It specifies the number of pixels in the plot view to increment by when calculating the next point. The default value is 1. Increasing this value will result in higher performance since fewer points need to be calculated but less accuracy in the plot.
5. The position of the X/Y axes, specified as a complex number. The default value is (0,0).

When executed, this operation pushes the current value of PPAR onto the stack. If the PPAR variable is not set, an undefined name error is raised.

# PREDV

**Member Of Menu:** Stat
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Symbol
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation predicts a value for the dependent variable given an input value from the independent variable by using the results from the most recent linear regression. It expects to find a variable called ∑PAR which is a list of four real values. The third and fourth real value in that list is the intercept and slope of the line returned by the linear regression. The intercept and slope are best calculated by using the LR operation.

The operation calculates the predicted value using the formula:

$Real_2$ = ( $Real_1$ x slope) + intercept

# PURGE

**Calculator Key:** ▪Purge

**Member Of Menu:** None
**Argument Types:** List
Symbol
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Symbol_1$ | → |
| $List_1$ | → |

This operation is used to remove a symbol from the symbol table, deleting the value associated with it. The operation either takes a single symbol from the top of the stack or a list of one or more symbols. It only attempts to delete the symbol from the current directory and does not traverse parent directories.

# PUT

**Member Of Menu:** List
Array
**Argument Types:** Any
Real
Complex
List
Symbol
Array
**Result Type(s):** None
List
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | | |
|---|---|---|---|---|
| $List_1$ | $Real_2$ | $Item_3$ | → | $List_4$ |
| $Array_1$ | $Real_2$ | $Item_3$ | → | $Array_4$ |
| $Array_1$ | $List_2$ | $Item_3$ | → | $Array_4$ |
| $Symbol_1$ | $Real_2$ | $Item_3$ | → | |
| $Symbol_1$ | $List_2$ | $Item_3$ | → | |

This operation is used to replace an item in a list, vector or matrix at a particular index. It can either modify an item on the stack or a value stored in memory, referenced by a symbol name. If provided a list on the stack, the real number is used as an index into the list to find the item to replace and the new item to put at that location is found at the top of the stack. The index for a matrix can also be a list with a single real number. The modified list is pushed onto the stack.

Similarly, the operation can modify a vector. The index for a vector can be a real number or a list with a single real number. The item to put in the vector must be a real or complex value. If the item to put in the vector is complex, the vector itself must be complex.

Finally, the operation can modify a matrix. The index for a matrix must be a list with two real values representing the row number of column number of the item. The item to put in the matrix must be a real or complex value. If the item to put in the matrix is complex, the matrix itself must be complex.

If provided a symbol, the value of that symbol is retrieved which should be a list, vector or matrix. The item at that index is replaced with the new item and then stored as the value of that symbol. In that case, nothing is returned by the operation.

# PUTI

**Member Of Menu:** List
Array
**Argument Types:** Any
Real
Complex
List
Symbol
Array
**Result Type(s):** Real
List
Symbol
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | | | |
|---|---|---|---|---|---|
| $List_1$ | $Real_2$ | $Item_3$ | → | $List_4$ | $Real_5$ |
| $List_1$ | $List_2$ | $Item_3$ | → | $List_4$ | $List_5$ |
| $Array_1$ | $Real_2$ | $Item_3$ | → | $Array_4$ | $Real_5$ |
| $Array_1$ | $List_2$ | $Item_3$ | → | $Array_4$ | $List_5$ |
| $Symbol_1$ | $Real_2$ | $Item_3$ | → | $Symbol_1$ | $Real_4$ |
| $Symbol_1$ | $List_2$ | $Item_3$ | → | $Symbol_1$ | $List_4$ |

This operation is used to replace an item in a list, vector or matrix at a particular index. It can either modify a list, vector or matrix on the stack or a one stored in memory, referenced by a symbol name. If provided a list, the real number is used as an index into the list to find the item to replace and the new item to put at that location is found at the top of the stack. The modified list is pushed onto the stack. Also, the index is incremented and also pushed onto the stack. The intention is to then push a new item onto the stack for the next item on the list and execute PUTI again. This makes it easy to quickly modify all items on the list, one after the other. For a list, the index can also be a list with a single real value.

Similarly, the operation can modify a vector. The index for a vector can be a real number or a list with a single real number. The item to put in the vector must be a real or complex value. If the item to put in the vector is complex, the vector itself must be complex.

Finally, the operation can modify a matrix. The index for a matrix must be a list with two real values representing the row number of column number of the item. The item to put in the matrix must be a real or complex value. If the item to put in the matrix is complex, the matrix itself must be complex. When incrementing the index on a matrix, the operation will visit each value in a row, column by column before proceeding to the next row.

If provided a symbol, the value of that symbol is retrieved which should be a list, vector or matrix. The item at that index is replaced with the new item and then stored as the value of that symbol. The symbol is left on the stack and the index is incremented so it points to the next index. Again,

this allows all items in the list to be quickly modified, one after the other.

When incrementing the index, if the next value is beyond the end of the list, vector or matrix, the index wraps to the first item. Also, this operation sets the 46th flag as described in the Working With Programs guide if the index wrapped. Otherwise, it clears the flag. Testing this flag can be useful in a program in order to loop over all items in the list, vector or matrix.

# P→R

**Member Of Menu:** Trig
Complex
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function takes a complex value expressed in polar coordinates, a radius and an angle, and returns a complex value expressed in rectangular coordinates. Note that if a real value is passed into this function, that same real value is returned.

# QUAD

**Member Of Menu:** Solv
Algebra
**Argument Types:** Symbol
Expression
**Result Type(s):** Expression
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$\text{Expression}_1$ $\text{Symbol}_2$ → $\text{Expression}_3$

This operation takes an expression and a symbol and finds the root of the expression using the quadratic equation. The expression must be a polynomial of the symbol to solve for. The degree of the polynomial can be greater than two but those higher order terms are discarded. So, the result for a polynomial greater than two is not actually a good solution for the roots of the polynomial. But for actual quadratic equations, the result can be used to find the exact roots of the expression.

The result will also have a "s1", "s2", etc variable in the expression. The actual name of the variable is unique so the number is chosen to make it unique. This represents the positive and negative roots in the quadratic equation. Set the sign variable to one to evaluate the positive root and minus one to evaluate the negative root.

If the expression 'A*SQ(X)+B*X+C' is solved for the symbol X using this operation, the result will be '(-B+s1*√(SQ(B)-4*A*C))/(2*A)'. This is the standard definition of the quadratic equation.

Before the expression is pushed onto the stack, it is evaluated recursively which means any symbols in the expression which have values associated with them will be replaced with their values. If the value is itself an expression of other symbols, they also will be replaced with their values.

The result can then be used with STEQ and the SOLVR to quickly evaluate the two roots or evaluate for other unknowns remaining in the equation.

# RAD

**Member Of Menu:** Mode
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation puts the calculator in radians mode. What this means is that any other operations which operates on angles (like trig functions), the angle is assumed to be expressed in radians. Also, operations which return an angle will return an angle expressed in radians. Note that if the input or output value is complex, it is always assumed to be expressed in radians, regardless of whether radian mode is on or not.

# RAND

**Member Of Menu:** Real
**Argument Types:** None
**Result Type(s):** Real
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| → | $Real_1$ |

This operation takes no input parameters and returns a random real valued result. The value is greater than or equal to 0 and strictly less than 1.

# RCEQ

**Member Of Menu:** Solv
Plot
**Argument Types:** None
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| → | $Item_1$ |

This operation looks up the symbol "EQ" in the current directory or one of its parents and pushes the value of that symbol onto the stack. Normally, an equation is what should be stored in that symbol.

The EQ symbol is looked up by the SOLVR operation and is the equation which is used when in that mode. Also, this equation is used when plotting a graph.

# RCL

**Calculator Key:** ▪Rcl

**Member Of Menu:** None
**Argument Types:** Symbol
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Symbol_1$ | → | $Item_2$ |

This operation is used to get the value of a symbol. It takes a symbol from the top of the stack and pushes the value associated with that symbol. The symbol is searched in the current directory and if it is not found, it will check the parent directory. This continues until it is found or it reaches the HOME directory and it is unable to find it. If unable to find the symbol when traversing the directories, an error is returned.

# RCLF

**Member Of Menu:** Test
**Argument Types:** None
**Result Type(s):** Integer
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

→ $\text{Integer}_1$

This operation retrieves the current calculator flags and pushes it as an integer value onto the stack. See this page for information about the calculator flags.

Note that you should make sure to set the integer word size to 64 (see the STWS for more information) before working with the flags. Otherwise, integer values on the stack will be truncated to fewer than 64 bits and you will not end up seeing all bits in the calculator flags.

# RCL∑

**Member Of Menu:** Stat
**Argument Types:** None
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| → | $Any_1$ |

This operation pushes the value stored in the ∑DAT variable onto the stack. If the ∑DAT variable does not exist, then this operation fails.

Although it is possible that any type of value could be stored in and retrieved from the ∑DAT variable, most stats operations expects a real matrix to be in that variable.

# RCWS

**Member Of Menu:** Binary
**Argument Types:** None
**Result Type(s):** Real
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

→ $Real_1$

This operation gets the word size of all integer values on the calculator. It pushes a real value to the stack which is the current word size for integers. The word size defaults to 64 bits.

All integers entered or operated on by other functions are masked to the word size of the calculator.

# RDM

**Member Of Menu:** Array
**Argument Types:** List
Symbol
Array
**Result Type(s):** None
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Array_1$ | $List_2$ | → | $Array_3$ |
| $Symbol_1$ | $List_2$ | → | |

This operation re-dimensions an input vector or matrix according to the size specified by the list. If the list has a single real value, the result will be a vector with that many values. If the list has two real values, the result will be a matrix with that many rows and columns.

The input vector or matrix can come from the stack or from a symbol. If the input is a symbol, then the value of that symbol most be a vector or matrix. The result will be push onto the stack or stored into the symbol if the input was a symbol.

The input vector or matrix is used to supply values for the output re-dimensioned array. If the input array does not have a value at a location in the output, then a zero appears there. So, if you shrink an array in a dimension, those values are lost on the output. If you grow an array in a dimension, the new values are set to zero.

# RDX,

**Member Of Menu:** Mode
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation switches between the "." and the "," character to represent the radix point in numerical output. By default, this setting is derived from the locale conventions set on the device but it can be changed to deviate from those conventions using this operation.

When set, the "," character separates the integer and fractional part of a number while the "." character is used to separate values (for example between the real and imaginary part of a complex number). When unset, the "." character separates the integer and fractional part of a number while the "," character is used to separate values.

# RDZ

**Member Of Menu:** Real
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Real_1$ | → |

This function takes a real valued random seed for use in future random numbers generated from RAND.

# RE

**Member Of Menu:** Complex
Array
**Argument Types:** Real
Complex
Symbol
Expression
Array
**Result Type(s):** Real
Expression
Array
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_1$ |
| $Complex_1$ | → | $Real_2$ |
| $Array_1$ | → | $Array_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function takes a complex value as input and returns the real component of that complex number. If the input value is a real value, then that real value is returned unchanged.

This operation also works with vectors and matrices. The result is a vector or matrix with the same dimensions of the input value. If the input is a complex valued vector or matrix, the result is a real valued vector or matrix consisting of the real component of each input value. If the input is a real valued vector or matrix, the result is the same real vector or matrix.

# REPEAT

**Member Of Menu:** Branch
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Real_1$ →

This operation is used in conjunction with the WHILE operation. See that page for more details.

# RES

**Member Of Menu:** Plot
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Real_1$ →

This operation takes a real number from the top of the stack and interprets that number as the "resolution" to use in a future plot. It stores that information in the plot parameters, found in a variable called PPAR in the current directory. The existing resolution which may be specified in an existing PPAR variable will be changed and then PPAR is stored, replacing the old value. If PPAR did not exist prior to executing these operations, a new PPAR variable is created. Its contents will be defaults except for the resolution which is specified from the stack.

The default resolution is 1 which means that each pixel in the plot view should be calculated when plotting an equation. Increasing this value to 2 will cause the calculator to evaluate the equation for every other pixel. Higher values reduce the resolution even further but increase the performance of the plot.

# RL

**Member Of Menu:** Binary
**Argument Types:** Integer
Symbol
Expression
**Result Type(s):** Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Integer_1$ | → | $Integer_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes an integer from the stack, shifts each bit one position to the left. The upper bit (which depends on the current word size of the calculator) is rotated and becomes the 0th bit. Assuming a 4 bit word size, then the binary number 1001 will become 0011 after this operation.

# RLB

**Member Of Menu:** Binary
**Argument Types:** Integer
Symbol
Expression
**Result Type(s):** Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Integer_1$ | → | $Integer_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes an integer from the stack, shifts each byte one position to the left. The upper byte (which depends on the current word size of the calculator) is rotated and becomes the lower byte. Assuming a 16 bit word size, then the hexadecimal number 1234 will become 2341 after this operation.

# RND

**Member Of Menu:** Real
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

Given the formatting mode for numbers, this function rounds those numbers to the digits displayed, removing any digits which are not displayed. If using FIX, SCI or ENG formatting mode, then the number of significant digits specified in that mode determines the rounding which occurs with this function.

# RNRM

**Member Of Menu:** Array
**Argument Types:** Array
**Result Type(s):** Real
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Array_1$ | → | $Real_2$ |

This operation calculates the row norm or infinity norm of the input vector or matrix.

# ROLL

**Calculator Key:** ▪Roll

**Member Of Menu:** None
**Argument Types:** Any
Real
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$$Item_1 \quad Item_2 \quad ... \quad Item_{n-1} \quad Item_n \quad Real_{n+1} \rightarrow Item_2 \quad ... \quad Item_{n-1} \quad Item_n \quad Item_1$$

This operation takes a real number from the stack which is the number of items on the stack to roll down. That number is referred to as "n" in the rest of this description. The item "n" items below that real number on the stack is then removed and pushed onto the top of the stack, rolling those "n" items.

# ROLLD

**Member Of Menu:** Stack
**Argument Types:** Any
Real
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Item_1$ $Item_2$ ... $Item_{n-1}$ $Item_n$ $Real_{n+1}$ → $Item_n$ $Item_1$ $Item_2$ ... $Item_{n-1}$

This operation takes a real number from the stack which is the number of items on the stack to roll down. The item below that real number on the stack is then removed and inserted "n" items from the top of the stack according to the value of that real number.

# ROOT

**Member Of Menu:** Solv
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | | |
|---|---|---|---|---|
| $\text{Expression}_1$ | $\text{Symbol}_1$ | $\text{Real}_3$ | → | $\text{Real}_4$ |
| $\text{Expression}_1$ | $\text{Symbol}_1$ | $\text{Complex}_3$ | → | $\text{Real}_4$ |
| $\text{Expression}_1$ | $\text{Symbol}_1$ | $\text{List}_3$ | → | $\text{Real}_4$ |

This operation takes an expression, a symbol to solve for and a real, complex or list of guesses and produces a root for the symbol in the expression. For an expression without an equal sign, a root for a variable is a real value for that variable for which the expression evaluates to zero. If the expression contains an "=" operation, it is treated like a subtraction and that expression is then used to search for a root.

The guess should be a value which you believe to be near a root. Alternatively, the guess can be a complex value. In this case, the real value of that complex value is used as the guess and the imaginary component is ignored. Also, a list of one, two or three real or complex values can be used as the guess. This allows multiple guesses which can help the root finding algorithm to find the appropriate root. Ideally, the guesses will be on either side of the actual root.

It is possible for the root finding algorithm to fail. In some cases, it will produce a warning that a root could not be found. Or, it may end up searching for the root forever. If after a reasonable amount of time, the operation does not complete, use the "Attn" button to interrupt the search.

# ROT

**Member Of Menu:** Stack
**Argument Types:** Any
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Item_1$ $Item_2$ $Item_3$ → $Item_2$ $Item_3$ $Item_1$

This operation rotates the top three items on the stack. In the stack diagram, $Item_3$ was at the top of the stack but after the operation completes, $Item_1$ is now at the top of the stack.

# RR

**Member Of Menu:** Binary
**Argument Types:** Integer
Symbol
Expression
**Result Type(s):** Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Integer_1$ | → | $Integer_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes an integer from the stack, shifts each bit one position to the right. The lower bit is rotated and becomes the upper bit (which depends on the current word size of the calculator). Assuming a 4 bit word size, then the binary number 1001 will become 1100 after this operation.

# RRB

**Member Of Menu:** Binary
**Argument Types:** Integer
Symbol
Expression
**Result Type(s):** Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Integer_1$ | → | $Integer_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes an integer from the stack, shifts each byte one position to the right. The lower byte is rotated and becomes the upper byte (which depends on the current word size of the calculator). Assuming a 16 bit word size, then the hexadecimal number 1234 will become 4123 after this operation.

# RSD

**Member Of Menu:** Array
**Argument Types:** Array
**Result Type(s):** Array
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

$\text{Array}_1$ $\text{Array}_2$ $\text{Array}_3$ → $\text{Array}_4$

This operation calculates the residual of its inputs which is:

$$\text{Array}_1 - \text{Array}_2 \times \text{Array}_3$$

If $\text{Array}_3$ is an approximation to a solution of $\text{Array}_2 \times X = \text{Array}_1$, then this provides a correction to improve that solution.

There are many requirements for the input values:

- $\text{Array}_1$ and $\text{Array}_3$ must either both be vectors or both be matrices.
- If $\text{Array}_1$ and $\text{Array}_3$ are both matrices, they must have the same number of columns.
- If $\text{Array}_1$ is a matrix, then $\text{Array}_1$ must have the same number of rows as in $\text{Array}_2$.
- If $\text{Array}_1$ is a vector, then the number of values in $\text{Array}_1$ must be the same as the number of rows in $\text{Array}_2$.
- $\text{Array}_2$ must be a matrix.
- If $\text{Array}_3$ is a matrix, then the number of rows of $\text{Array}_3$ must be the same as the number of columns of $\text{Array}_2$.
- If $\text{Array}_3$ is a vector, then the number of values of $\text{Array}_3$ must be the same as the number of columns of $\text{Array}_2$.

# R→B

**Member Of Menu:** Binary
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Integer_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes a real number and converts it to an integer value. If the real value is less than 0, then the result is an integer 0. Otherwise, the real value is rounded to the nearest integer and pushed to the stack as an integer value. Note that the the word size may result in upper bits being masked out of the result.

# R→C

**Member Of Menu:** Trig
Complex
Array
**Argument Types:** Real
Symbol
Expression
Array
**Result Type(s):** Complex
Expression
Array
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | ($Real_1$, $Real_2$) |
| $Array_1$ | $Array_2$ | → | $Array_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This function takes two real values and returns a complex value. One real value is used as the real component of the complex result while the other real value is used as the imaginary component.

This operation can also take two real valued vectors or matrices and produce a complex valued vector or matrix. The two input arrays must be the same dimension. They must both be vectors or both be matrices. The input arrays must be real valued. The result will be a complex array where the real component of each value comes from $Array_1$ and the imaginary component of each value comes from $Array_2$.

# R→D

**Member Of Menu:** Trig
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function takes a real number which is an angle expressed in radians and converts it to an angle expressed in degrees.

# R→P

**Member Of Menu:** Trig
Complex
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function takes a complex value expressed in rectangular coordinates and returns a complex value expressed in polar coordinates, a radius and an angle. Note that if a real value is passed into this function, that same real value is returned.

# SAME

**Member Of Menu:** Test
**Argument Types:** Any
**Result Type(s):** Real
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Item_1$ $Item_2$ → $Real_3$

This operation takes two items from the stack and pushes a real value which is 1 if the two items are the same, otherwise a 0 is pushed onto the stack. An item is the same if it is the same type and the same value.

Unlike the $\underline{==}$ operation, this operation can be used to compare symbols and expressions. The == operation does not compare an expression itself and instead will form a new expression which will be evaluated when the expression is evaluated. So, SAME is the only way to compare whether two expressions are identical or not. Note that an expressions must match exactly to be considered the same so 'x + y' is not the same as 'y + x' although they are effectively the same.

# SCI

**Member Of Menu:** Mode
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Real_1$ | → |

This operation enables the scientific format for real numbers on the calculator. The scientific format takes a real argument which is the number of digits to display to the right of the decimal point. When enabled, there will always be that many digits shown to the right of the decimal place, even if they are all 0's. Also, exponential format is always used in scientific format.

# SCONJ

**Member Of Menu:** Store
**Argument Types:** Symbol
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $\text{Symbol}_1$ | → |

This operation takes a symbol from the stack. It looks for the value of that symbol in the current directory, calculates the CONJ of the value, then stores that result back to that symbol. The old value of the symbol is replaced with the new calculated value.

# SDEV

**Member Of Menu:** Stat
**Argument Types:** None
**Result Type(s):** Real
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| → | $Real_1$ |
| → | $Array_1$ |

This operation calculates the standard deviation of the statistics data. It expects to find a variable called ∑DAT which has a set of statistics values stored in a real matrix. Each row in the real matrix represents a single set of samples. Each column contains one of the set of values associated with a single sample. The operation calculates the standard deviation of each column of data in the real matrix and pushes the result into the stack.

If the real matrix has a single column, then a real value which is the standard deviation of all values in that column is pushed onto the stack. If the real matrix has two or more columns, then a real vector with the same number of columns is pushed onto the stack where each value in the vector is the standard deviation of values from that column.

Note that the ∑DAT must have at least two rows in order to calculate a standard deviation.

# SF

**Member Of Menu:** Test
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Real_1$ →

This operation takes a real value between 1 and 64 and sets the associated flag bit to one. See this page for information about the calculator flags.

# SHOW

**Member Of Menu:** Solv
Algebra
**Argument Types:** Symbol
Expression
**Result Type(s):** Expression
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$\text{Expression}_1 \ \ \text{Symbol}_2 \ \rightarrow \ \text{Expression}_3$

This operation takes an expression and a target symbol. It then recursively evaluates the value of each symbol in the expression to see if that symbol can be replaced with an expression which contains the target symbol. If the symbol in the expression can be be expressed in terms of the target symbol, then it is left unchanged.

As an example, if the symbol X has value 'A+B' and Y has value 'C+D', then if SHOW is executed on the expression 'X+Y' with a target symbol B, the result will be 'A+B+Y'.

# SIGN

**Member Of Menu:** Complex
Real
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

If given a positive real value, this function returns 1. If the real value is 0, this function returns 0. If this real value is negative, this function returns -1.

For complex inputs, this function returns a complex value with the same ARG (ie angle) but its ABS is 1. If you treat the complex value as a vector from the origin, then the result points in the same direction but the length of the resulting vector is 1, regardless of the length of the input vector. The exception is if the input complex value is (0, 0). In that case, the result is also (0, 0).

# SIN

**Member Of Menu:** Trig
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the sine function of its argument. Note that the result depends on the whether the DEG (degree) or RAD (radians) mode is on. For real valued arguments, the argument is treated as an angle in degrees if DEG is on. For real valued arguments, the argument is treated as an angle in radians if RAD is on. For complex arguments, the angle is always expected to be in radians.

# SINH

**Member Of Menu:** Logs
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the hyperbolic sine function of its input argument. It takes real or complex input values.

# SINV

**Member Of Menu:** Store
**Argument Types:** Symbol
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Symbol_1$ | → |

This operation takes a symbol from the stack. It looks for the value of that symbol in the current directory, calculates the INV of the value, then stores that result back to that symbol. The old value of the symbol is replaced with the new calculated value.

# SIZE

**Member Of Menu:** String
List
Array
Algebra
**Argument Types:** List
String
Symbol
Expression
Array
**Result Type(s):** Real
List
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $String_1$ | → | $Real_2$ |
| $List_1$ | → | $Real_2$ |
| $Vector_1$ | → | $Real_2$ |
| $Matrix_1$ | → | $List_2$ |
| $Symbol_1$ | → | $Real_2$ |
| $Expression_1$ | → | $Real_2$ |

This operation takes a string, list, vector, matrix or expression argument. In the case of a string, it pushes the number of characters in the string onto the stack. For a list argument, it pushes the number of items in the list onto the stack. For a vector it pushes a real number representing the number of values in the vector. For a matrix, it pushes a list of two real numbers representing the number of rows and the number of columns in the matrix. For an expression, it pushes the number of components in the expression. In other words, the number of real numbers, complex numbers, symbols and operations in the expression.

# SL

**Member Of Menu:** Binary
**Argument Types:** Integer
Symbol
Expression
**Result Type(s):** Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Integer_1$ | → | $Integer_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes an integer from the stack, shifts each bit one position to the left. The upper bit (which depends on the current word size of the calculator) is lost and a zero is shifted into the 0th bit position. Assuming a 4 bit word size, then the binary number 1001 will become 0010 after this operation.

# SLB

**Member Of Menu:** Binary
**Argument Types:** Integer
Symbol
Expression
**Result Type(s):** Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Integer_1$ | → | $Integer_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes an integer from the stack, shifts each byte one position to the left. The upper byte (which depends on the current word size of the calculator) is lost and 0 is shifted into the lower byte. Assuming a 16 bit word size, then the hexadecimal number 1234 will become 2340 after this operation.

# SNEG

**Member Of Menu:** Store
**Argument Types:** Symbol
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Symbol_1$ | → |

This operation takes a symbol from the stack. It looks for the value of that symbol in the current directory, calculates the NEG of the value, then stores that result back to that symbol. The old value of the symbol is replaced with the new calculated value.

# SOLVR

**Member Of Menu:** Solv
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation gets the value of EQ from the current directory or one of its parent and goes into a special mode where an equation stored in that symbol can be easily evaluated. Normally, the value of EQ should be an expression.

After executing this operation, the "Solvr" menu buttons is opened. The buttons which appear in this menu depends on the expression being manipulated. The set of symbols which appear in the expression are represented by buttons in the menu. However, if one of these symbols has a value associated with it which is itself an expression, it is then replaced with the symbols which appear in that expression.

As an example, imagine that the expression 'SIN(SQ(X)+Y)' is being manipulated. In this case, the symbols X and Y will appear as buttons in the menu. If the value of X is then set to 'A+B', the X button will disappear from the menu and two new buttons for A and B will instead appear. Also, if the value of X is then cleared, the A and B buttons will be removed and X will reappear.

If the expression does not contain a "=" operation, then a button will appear with the label "EXPR=". Otherwise, the buttons "LEFT=" and "RIGHT=" will appear.

Commonly, in this mode you will push values for the variables onto the stack. If a button for the variable X appears in the menu and you want to set X to 5, push 5 onto the stack and press the X button. The variable X will be set to that value from the stack. With values set for all variables, you can then press the "EXPR=" button to evaluate the expression for those variables. Alternatively if the expression contains a "=" operation, you can press the "LEFT=" or "RIGHT=" to evaluate the left or right side of the expression.

A root for a variable can also be searched in this mode. For an expression without an equal sign, a root for a variable is a real value for that variable for which the expression evaluates to zero. If the expression contains an "=" operation, it is treated like a subtraction and that expression is then used to search for a root.

To find a root for an expression of X, first set X to a value which you believe to be near a root. Then, press ▪X. If the expression 'SQ(X)-2' is being manipulated, then the root we are searching for is square root of 2. This can be found by setting X to 2 which will be our guess where the root might be and then pressing ▪X. After a short time, the root is pushed onto the stack and the calculator will pop up a notice to say it found a root. In this case, 1.41421356237 is pushed onto the stack. Also, X will have that value now.

Alternatively, you can set X to a complex value as your guess for the root. In this case, the real value of that complex value is used as the guess and the imaginary component is ignored. Also, a list of one, two or three real or complex values can be stored in the variable as our guess. This allows multiple guesses which can help the root finding algorithm to find the appropriate root. Ideally, the guesses will be on either side of the actual root.

In some cases, the algorithm may never find the root. In this case, press the "Attn" button on the calculator to interrupt its search.

If you set the variable to a list of three values, the search may result in finding a maximum or minimum value of the expression. To search for a maximum value, the lowest and highest guess must be on either side of the maximum and the middle guess must be closer to the maximum than the other two guesses. Similarly, the three guesses must contain the minimum. If the maximum or minimum is not in the range of the guesses, then the guesses will instead be used to find search for a root. When a max or min is found, the location of the maximum or minimum is pushed onto the stack and stored into the variable. Also, the calculator pops up a message to say what it found.

# SQ

**Calculator Key:** ■$x^2$

**Member Of Menu:** None
**Argument Types:** Real
Complex
Symbol
Expression
Array
**Result Type(s):** Real
Complex
Expression
Array
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Array_1$ | → | $Array_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

The SQ operation multiplies its input argument by itself to produce the square of that input value. The value can be a real value, complex value or square matrix. The operation will fail on a non-square matrix or a vector. The result is the same as the input argument multiplied by itself.

# SR

**Member Of Menu:** Binary
**Argument Types:** Integer
Symbol
Expression
**Result Type(s):** Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Integer_1$ | → | $Integer_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes an integer from the stack, shifts each bit one position to the right. The lower bit is lost and a zero is shifted into the upper bit position (which depends on the current word size of the calculator). Assuming a 4 bit word size, then the binary number 1001 will become 0100 after this operation.

# SRB

**Member Of Menu:** Binary
**Argument Types:** Integer
Symbol
Expression
**Result Type(s):** Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Integer_1$ | → | $Integer_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes an integer from the stack, shifts each byte one position to the right. The lower byte is lost and a zero is shifted into the upper byte (which depends on the current word size of the calculator). Assuming a 16 bit word size, then the hexadecimal number 1234 will become 0123 after this operation.

# SST

**Member Of Menu:** Control
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation will single step the currently HALT-ed program. By placing a HALT at an opportune location in a program, you can then use SST to single step through the program, inspecting the stack as it changes. At any point you can continue or abort execution of the current program.

# START

**Member Of Menu:** Branch
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | $Real_2$ | → |

The START operation is used to define a loop structure within a program. It is combined with the NEXT or STEP operation to define the boundaries of the loop.

The normal way START is used is one of the following:

```
« ... Real1 Real2 START ... operations ... NEXT ... »
```

or

```
« ... Real1 Real2 START ... operations ... Real3 STEP ... »
```

When START is executed, it pops two values of the stack which should be real values. $Real_1$ is the starting value for the loop counter and $Real_2$ is the ending value for the loop counter. When NEXT is executed, the loop counter is incremented by one. If the loop counter is less than or equal to the ending value, then execution jumps back to the operation following START and executes the body of the loop again. If the value is greater than the ending value after incrementing the loop counter, then execution continues after the NEXT operation, exiting the loop.

The STEP operation can be used in place of NEXT to specify an increment for the loop counter which is not one. When STEP is executed the top of the stack is popped. It expects to find a real value there. It adds that real value to the loop counter and then tests to see if the loop counter has reached the end. For a negative increment, the loop stops when the counter is less than the ending value. For a positive increment, the loop stops when the counter is greater than the ending value. If not yet at the end, it loops back to the operation following START. Otherwise, it exits the loop and execution continues after the STEP operation.

Note that in the body of the loop you do not have access to the loop counter. If you need access to the loop counter, you may want to use the FOR loop instead.

An error will be raised if this operation is used outside of program execution context.

# STD

**Member Of Menu:** Mode
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation enables the standard format for real numbers on the calculator. In standard format, a number will appear without an exponent if it can be represented in 12 or fewer digits. If more digits are required for 0's to the left or the right of the decimal point, then exponential format will be used. In exponential format, numbers look like 1.111E13 or 2.222E-12 which means 1.111 x $10^{13}$ and 2.222 x $10^{-12}$ respectively.

# STEP

**Member Of Menu:** Branch
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Real_1$ →

This operation is used in conjunction with the START or FOR operations. See those pages for more details.

# STEQ

**Member Of Menu:** Solv
Plot
**Argument Types:** Any
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Item_1$ | → |

This operation takes any item from the stack and stores it in the current directory under the symbol "EQ". Normally, an equation is what should be stored into that symbol.

The EQ symbol is looked up by the SOLVR operation and is the equation which is used when in that mode. Also, this equation is used when plotting a graph.

# STO

**Calculator Key:** Sto

**Member Of Menu:** None
**Argument Types:** Any
Symbol
Expression
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Item_1$ | $Symbol_2$ | → |
| $Item_1$ | $Expression_2$ | → |

This operation is used to store an item on the stack into a named symbol. Anything which can be put onto the stack can be stored. The top of the stack is expected to be a symbol (for example 'X' to store to the X variable). Below that is the value to store in that symbol. After execution, both these values are popped from the stack. The symbol is created in the current directory.

Also, it is possible to change a particular item in a list, vector or matrix using this operation. If the variable X has a value which is a list, then you can change the second item in that list by pushing the new item and then 'X(2)' to the stack. When executing this operation, it treats the expression like a symbol and an index into the list associated with that symbol. If the value is not a list or the index is out of the bounds of the list, an error is returned. In this case, the symbol is expected to be found in the current directory and that value is the one updated.

Similarly, if X is a vector, then storing a value into 'X(2)' will change the second value in the vector. The value being stored must be a real or complex value. Finally, if X is a matrix, then storing a value into 'X(2,3)' will change the value at the second row, third column. Again the value being store must be real or complex.

# STO*

**Member Of Menu:** Store
**Argument Types:** Real
Complex
Integer
Symbol
Array
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Symbol_1$ | $Real_2$ | → |
| $Symbol_1$ | $Complex_2$ | → |
| $Symbol_1$ | $Integer_2$ | → |
| $Symbol_1$ | $Array_2$ | → |
| $Real_1$ | $Symbol_2$ | → |
| $Complex_1$ | $Symbol_2$ | → |
| $Integer_1$ | $Symbol_2$ | → |
| $Array_1$ | $Symbol_2$ | → |

This operation takes a symbol and another item of just about any numeric type (real, complex, integer or matrix). The symbol can be in either position on the stack and the other item on the stack must be a numeric input. It gets the value of that symbol from the current directory and multiplies that value with the other numeric item from the stack. It then stores the result back to the symbol. The old value of the symbol is replaced with the new calculated value. Nothing is returned on the stack because the result is stored in the symbol.

# STO+

**Member Of Menu:** Store
**Argument Types:** Real
Complex
Integer
Symbol
Array
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Symbol_1$ | $Real_2$ | → |
| $Symbol_1$ | $Complex_2$ | → |
| $Symbol_1$ | $Integer_2$ | → |
| $Symbol_1$ | $Array_2$ | → |
| $Real_1$ | $Symbol_2$ | → |
| $Complex_1$ | $Symbol_2$ | → |
| $Integer_1$ | $Symbol_2$ | → |
| $Array_1$ | $Symbol_2$ | → |

This operation takes a symbol and another item of just about any numeric type (real, complex, integer or matrix). The symbol can be in either position on the stack and the other item on the stack must be a numeric input. It gets the value of that symbol from the current directory and adds that value with the other numeric item from the stack. It then stores the result back to the symbol. The old value of the symbol is replaced with the new calculated value. Nothing is returned on the stack because the result is stored in the symbol.

# STO-

**Member Of Menu:** Store
**Argument Types:** Real
Complex
Integer
Symbol
Array
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Symbol_1$ | $Real_2$ | → |
| $Symbol_1$ | $Complex_2$ | → |
| $Symbol_1$ | $Integer_2$ | → |
| $Symbol_1$ | $Array_2$ | → |
| $Real_1$ | $Symbol_2$ | → |
| $Complex_1$ | $Symbol_2$ | → |
| $Integer_1$ | $Symbol_2$ | → |
| $Array_1$ | $Symbol_2$ | → |

This operation takes a symbol and another item of just about any numeric type (real, complex, integer or matrix). The symbol can be in either position on the stack and the other item on the stack must be a numeric input. It gets the value of that symbol from the current directory and performs the subtract operation.

If the symbol is in stack level 1, then the value of the symbol is subtracted from the numeric argument from stack level 2. If the symbol is in stack level 2, then the numeric argument from stack level 1 is subtracted from the value of the symbol. Either way, the result of that operation is stored back to the symbol. The old value of the symbol is replaced with the new calculated value. Nothing is returned on the stack because the result is stored in the symbol.

# STO/

**Member Of Menu:** Store
**Argument Types:** Real
Complex
Integer
Symbol
Array
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Symbol_1$ | $Real_2$ | → |
| $Symbol_1$ | $Complex_2$ | → |
| $Symbol_1$ | $Integer_2$ | → |
| $Symbol_1$ | $Array_2$ | → |
| $Real_1$ | $Symbol_2$ | → |
| $Complex_1$ | $Symbol_2$ | → |
| $Integer_1$ | $Symbol_2$ | → |
| $Array_1$ | $Symbol_2$ | → |

This operation takes a symbol and another item of just about any numeric type (real, complex, integer or matrix). The symbol can be in either position on the stack and the other item on the stack must be a numeric input. It gets the value of that symbol from the current directory and performs the subtract operation.

If the symbol is in stack level 2, then the value of the symbol is divided by the numeric argument from stack level 1. If the symbol is in stack level 1, then the numeric argument from stack level 2 is divided by the value of the symbol. Either way, the result of that operation is stored back to the symbol. The old value of the symbol is replaced with the new calculated value. Nothing is returned on the stack because the result is stored in the symbol.

# STOF

**Member Of Menu:** Test
**Argument Types:** Integer
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $\text{Integer}_1$ | → |

This operation takes an integer value from the stack and sets the calculator flags to this value. See this page for information about the calculator flags.

Note that you should make sure to set the integer word size to 64 (see the STWS for more information) before working with the flags. Otherwise, integer values on the stack will be truncated to fewer than 64 bits and this will zero the upper bits of the calculator flags when they are set.

# STO∑

**Member Of Menu:** Stat
**Argument Types:** Any
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Any_1$ | → |

This operation takes the item from the top of the stack and stores it into a variable called ∑DAT. If the ∑DAT variable already contains a value, that value is overwritten by the value from the stack.

Although any value can be stored into the ∑DAT variable, it is expected that a real matrix will be stored into that variable. The stats operations expect a real matrix where every row represents a sample and each column contains one of the values from that sample. If you store anything other than a real matrix in the ∑DAT variable, most other stats operations will fail.

# STR→

**Member Of Menu:** String
**Argument Types:** String
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $String_1$ | → | $Item_2$ |

This operation takes a string from the stack and interprets its contents as though it was just entered and the resulting value, or values are pushed onto the stack. If the string cannot be parsed, an error is displayed.

# STWS

**Member Of Menu:** Binary
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Real_1$ | → |

This operation sets the word size of all integer values on the calculator. It takes a real value, rounds it to the nearest integer and uses that value to determine how many bits should be used to represent integers. The word size defaults to 64 bits. If the number is less than 1, the word size is set to 1. If the number is greater than 64, the word size is set to 64.

All integers entered or operated on by other functions are masked to the word size of the calculator.

# SUB

**Member Of Menu:** String
List
**Argument Types:** Real
List
String
**Result Type(s):** List
String
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | | |
|---|---|---|---|---|
| $String_1$ | $Real_2$ | $Real_3$ | → | $String_4$ |
| $List_1$ | $Real_2$ | $Real_3$ | → | $List_4$ |

This operation extracts a substring or a sub-list from a string or a list. If it is provided a string and two real indices, it will return a substring between those two indices. The first character in the string begins at index 1. If Real3 is less than Real2 then an empty string is returned.

Similarly, this operation can extract a subset of the items of a list. If Real3 is less than Real2 then an empty list is returned. Otherwise, the items between these two indices is returned. Note that the first item in the list is considered to be at index 1.

# SWAP

**Calculator Key:** ▪Swap

**Member Of Menu:** None
**Argument Types:** Any
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | | |
|---|---|---|---|---|
| $Item_1$ | $Item_2$ | → | $Item_2$ | $Item_1$ |

This operation pops the top two items off of the stack and pushes them back on in reverse order.

# TAN

**Member Of Menu:** Trig
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the tangent function of its argument. Note that the interpretation of the input argument depends on the whether the DEG (degree) or RAD (radians) mode is on. For real valued arguments, the argument is treated as an angle in degrees if DEG is on. For real valued arguments, the argument is treated as an angle in radians if RAD is on. For complex arguments, the angle is always expected to be in radians.

# TANH

**Member Of Menu:** Logs
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation calculates the hyperbolic tangent function of its input argument. It takes real or complex input values.

# TAYLR

**Member Of Menu:** Algebra
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Expression
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$\text{Expression}_1$ $\text{Symbol}_2$ $\text{Real}_4$ → $\text{Expression}_4$

This operation calculates an approximation of the Taylor series of the input expression at zero. The symbol argument indicates what symbol should be used to evaluate the Taylor series against. The real argument should be a positive integer value which is the degree of the Taylor series to create.

To successfully calculate the Taylor series, the expression and the first nth derivatives of that expression must have a real value at 0. If you want to evaluate an expression of X at some other value, for example at 3, then you could set X to the expression 'Y+3' and then evaluate your expression. This should substitute all X's in your expression with 'Y+3'. Then, you can get the Taylor series against Y. Finally, you can clear the value of X and set Y to 'X-3' and evaluate the result to get an expression in terms of X again.

# THEN

**Member Of Menu:** Branch
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation is used in conjunction with the IF or IFERR operations. See those pages for more details.

# TOT

**Member Of Menu:** Stat
**Argument Types:** None
**Result Type(s):** Real
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| → | $Real_1$ |
| → | $Array_1$ |

This operation calculates the total of the statistics data. It expects to find a variable called ∑DAT which has a set of statistics values stored in a real matrix. Each row in the real matrix represents a single set of samples. Each column contains one of the set of values associated with a single sample. The operation calculates the total of each column of data in the real matrix and pushes the result into the stack.

If the real matrix has a single column, then a real value which is the sum of all values in that column is pushed onto the stack. If the real matrix has two or more columns, then a real vector with the same number of columns is pushed onto the stack where each value in the vector is the sum of values from that column.

# TRN

**Member Of Menu:** Array
**Argument Types:** Symbol
Array
**Result Type(s):** None
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Array_1$ | → | $Array_2$ |
| $Symbol_1$ | → | |

This operation returns the transpose of its input array. The input array can be directly on the stack in which case the result is pushed onto the stack or it can be a symbol. If it is a symbol, the value of that symbol must be an array. No value is pushed onto the stack in this case and the value of the symbol is transposed.

Transpose cannot be performed on a vector. A transpose on a real matrix of N rows and M columns returns a real matrix of M rows and N columns. Item at row X, column Y appears at row Y column X in the resulting matrix.

For complex matrices, the same swapping of rows and columns occurs but each value is also set to its conjugate.

# TYPE

**Member Of Menu:** Test
**Argument Types:** Any
**Result Type(s):** Real
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Item_1$ → $Real_2$

This operation pops an item off the stack and then pushes a real number depending on the type of item it just popped off the stack. The following table describes the values you can expect to get for different types:

| Type | Value |
|---|---|
| Real | 0 |
| Complex | 1 |
| String | 2 |
| List | 5 |
| Global Symbol | 6 |
| Local Symbol | 7 |
| Program | 8 |
| Expression | 9 |
| Integer | 10 |

# UNDO

**Member Of Menu:** Mode
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation enables or disables the Undo functionality on the calculator. The Undo function allows you to recall the previous state of the stack after the most recent operation.

# UNTIL

**Member Of Menu:** Branch
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

This operation is used in conjunction with the DO operation. See that page for more details.

# UTPC

**Member Of Menu:** Stat
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation calculates the probability given a chi-square distribution. It returns the probability of a variable being greater than $Real_2$ with $Real_1$ degrees of freedom.

# UTPF

**Member Of Menu:** Stat
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | | |
|---|---|---|---|---|
| $Real_1$ | $Real_2$ | $Real_3$ | → | $Real_4$ |
| $Symbol_1$ | $Symbol_2$ | $Symbol_3$ | → | $Expression_4$ |
| $Expression_1$ | $Expression_2$ | $Expression_3$ | → | $Expression_4$ |

This operation calculates the probability given a F distribution. It returns the probability of a variable being greater than $Real_3$ given a distribution with degrees of freedom with numerator $Real_1$ and denominator $Real_2$.

# UTPN

**Member Of Menu:** Stat
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | | |
|---|---|---|---|---|
| $Real_1$ | $Real_2$ | $Real_3$ | → | $Real_4$ |
| $Symbol_1$ | $Symbol_2$ | $Symbol_3$ | → | $Expression_4$ |
| $Expression_1$ | $Expression_2$ | $Expression_3$ | → | $Expression_4$ |

This operation calculates the probability given a normal distribution. It returns the probability of a variable being greater than $Real_3$ in a normal distribution with mean $Real_1$ and variance $Real_2$.

# UTPT

**Member Of Menu:** Stat
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation calculates the probability given a t distribution. It returns the probability of a variable being greater than $Real_2$ in a t-distribution with $Real_1$ degrees of freedom.

# VAR

**Member Of Menu:** Stat
**Argument Types:** None
**Result Type(s):** Real
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| → | $Real_1$ |
| → | $Array_1$ |

This operation calculates the variance of the statistics data. It expects to find a variable called ∑DAT which has a set of statistics values stored in a real matrix. Each row in the real matrix represents a single set of samples. Each column contains one of the set of values associated with a single sample. The operation calculates the variance of each column of data in the real matrix and pushes the result into the stack.

If the real matrix has a single column, then a real value which is the variance of all values in that column is pushed onto the stack. If the real matrix has two or more columns, then a real vector with the same number of columns is pushed onto the stack where each value in the vector is the variance of values from that column.

Note that the ∑DAT must have at least two rows in order to calculate a variance.

# VARS

**Member Of Menu:** Memory
**Argument Types:** None
**Result Type(s):** List
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| → | $List_1$ |

This operation pushes a list onto the stack which contains the set of variables and directories in the current directory. The order of the symbols in the list matches the order of the symbols in the current directory.

Re-arranging the order of the items in this list is a great way to prepare to execute the ORDER operation.

# WAIT

**Member Of Menu:** Control
**Argument Types:** Real
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Real_1$ → 

This operation takes a real number which represents the number of seconds to pause before continuing execution. The number can be a whole number or can have fractional components in order to delay for fractions of seconds.

This operation is mostly useful within executing programs although can also be used outside of program execution context.

# WHILE

**Member Of Menu:** Branch
**Argument Types:** None
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No

The WHILE operation is used to define a loop structure within a program. It is combined with the REPEAT and END operation to define the boundaries of the loop.

The normal way WHILE is used is:

« ... WHILE ... operations ... REPEAT ... operations ... END ... »

The operations between WHILE and REPEAT are the test operations which determine whether to execute the loop operations and the operations between REPEAT and END are the loop operations. When REPEAT is reached, the top of the stack is popped. A real value is expected. If the real value is non-zero (true), then execution continues after the REPEAT operation, executing the loop operations and then once those operations are executed, execution loops back to the operation following WHILE. At that point, the test operations are re-evaulated. If the real value is zero (false), then execution continues after END and the loop terminates.

With a WHILE loop, the loop operations may not ever be executed. If the real value from the test operations is zero, then the loop operations are skipped and execution continues after END.

An error will be raised if this operation is used outside of program execution context.

# XOR

**Member Of Menu:** Binary
Test
**Argument Types:** Real
Integer
Symbol
Expression
**Result Type(s):** Real
Integer
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Integer_1$ | $Integer_2$ | → | $Integer_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation performs a binary xor operation on its two integer arguments and returns the integer result. If the input arguments are real values, it returns a 1 if only one of the two arguments is a 0, otherwise it returns 0.

Note that when used in an expression on symbols x and y for example, it would look like 'x XOR y'.

# XPON

**Member Of Menu:** Real
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function returns the exponent of the input real value. Assuming that the real value is expressed in scientific notation (the actual format being used on the calculator is irrelevant but for the purposes of explanation, assume the value is in scientific notation) like so:

x.xxxxEyy

Then, the value returned will be yy, returning only the exponent.

# ^

**Calculator Key:** ■^

**Member Of Menu:** None
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Real_1$ | $Real_2$ | → | $Complex_3$ |
| $Complex_1$ | $Complex_2$ | → | $Complex_3$ |
| $Real_1$ | $Complex_2$ | → | $Complex_3$ |
| $Complex_1$ | $Real_2$ | → | $Complex_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation calculates $x^y$ given that values x and y are pushed onto the stack in that order. Either argument can be real or complex and the result may be real or complex.

# ✖

**Calculator Key:** ×

**Member Of Menu:** None
**Argument Types:** Real
Complex
Integer
Symbol
Expression
Array
**Result Type(s):** Real
Complex
Integer
Expression
Array
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Complex_1$ | $Complex_2$ | → | $Complex_3$ |
| $Real_1$ | $Complex_2$ | → | $Complex_3$ |
| $Complex_1$ | $Real_2$ | → | $Complex_3$ |
| $Integer_1$ | $Integer_2$ | → | $Integer_3$ |
| $Real_1$ | $Integer_2$ | → | $Integer_3$ |
| $Integer_1$ | $Real_2$ | → | $Integer_3$ |
| $Real_1$ | $Array_2$ | → | $Array_3$ |
| $Array_1$ | $Real_2$ | → | $Array_3$ |
| $Complex_1$ | $Array_2$ | → | $Array_3$ |
| $Array_1$ | $Complex_2$ | → | $Array_3$ |
| $Array_1$ | $Array_2$ | → | $Array_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

The multiply operation will take its two numerical operands and produce the product as its result. It operates on reals, complex and integer values. Also, you can combine reals with complex values and reals with integer values. The result will be a complex or integer value respectively.

The multiply operation can also multiply a vector or matrix by a real or complex value. The result will be a vector or matrix with the same dimensions with each value from the input array

multiplied by the real or complex value.

The multiply operation can be used to multiply a matrix or vector with another matrix or vector. The following rules are enforced on the input parameters:

- $\text{Array}_1$ must not be a vector.
- $\text{Array}_2$ can be a vector. If it is, it is treated just like a matrix with one column and N rows where N is the number of values in the vector.
- The number of columns in $\text{Array}_1$ must match the number of rows in $\text{Array}_2$

The actual operation performed is a matrix multiplication.

The result will have the following properties:

- If $\text{Array}_2$ is a vector, then the result is a vector of the same dimension.
- Otherwise, the result will be a matrix with same number of rows as $\text{Array}_1$ and the same number of columns as $\text{Array}_2$.
- If any one input is a complex matrix or vector, then the result will be complex.
- Otherwise, the result will be real.

# ÷

**Calculator Key:** ÷

**Member Of Menu:** None
**Argument Types:** Real
Complex
Integer
Symbol
Expression
Array
**Result Type(s):** Real
Complex
Integer
Expression
Array
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Complex_1$ | $Complex_2$ | → | $Complex_3$ |
| $Real_1$ | $Complex_2$ | → | $Complex_3$ |
| $Complex_1$ | $Real_2$ | → | $Complex_3$ |
| $Integer_1$ | $Integer_2$ | → | $Integer_3$ |
| $Real_1$ | $Integer_2$ | → | $Integer_3$ |
| $Integer_1$ | $Real_2$ | → | $Integer_3$ |
| $Array_1$ | $Real_2$ | → | $Array_3$ |
| $Array_1$ | $Complex_2$ | → | $Array_3$ |
| $Array_1$ | $Array_2$ | → | $Array_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

The divide operation will take its two numerical operands and produce the quotient as its result. It operates on reals, complex and integer values. Also, you can combine reals with complex values and reals with integer values. The result will be a complex or integer value respectively.

Also, you can divide a matrix or vector by a real or complex value. The result will be a matrix or vector with the same dimension as the input with each value divided by the real or complex value.

If the inputs both arrays, then they must meet these criteria:

- $Array_2$ must be a square matrix
- $Array_1$ can be a vector. If it is a vector then it is treated just like a matrix with one column and N rows where N is the number of values in the vector.
- $Array_1$ and $Array_2$ must have the same number of rows.

The result will have the same dimensions as $Array_1$. Also, if $Array_1$ is a vector, the result will be a vector. Otherwise, it will be a matrix.

The result $Array_3$ is the solution to the equation which looks like this:

$$Array_2 \times Array_3 = Array_1$$

If a solution cannot be found, the calculator will present an infinite result error.

# →

**Calculator Key:** ■→

**Member Of Menu:** None
**Argument Types:** Any
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Item_1$ | ... | $Item_n$ | → |

This operation can only be used from within program context. In that context, it is used like this:

```
« ... → symbol1 ... symboln « anotherProgram » ...  »
```

or instead of a program, you can use an expression like this:

```
« ... → symbol1 ... symboln 'expression' ...  »
```

When the program executes, values "n" values will be popped off the stack. The value from the top of the stack will be stored in a local variable called $symbol_n$ and the last value popped off the stack will be stored in a local variable called $symbol_1$. Finally, the expression or program which follows the list of symbols will be executed with those local variables in its context. Once execution of that expression or program completes, the local variables are removed from the execution context.

If the → operation is at the beginning of the program and the expression/program it evaluates is the last thing in the program, then this program can be used as a custom operation. Store this program in a global variable. Assuming it is stored in a variable called CUSTOMOP, then it can be called from an expression like this:

```
'CUSTOMOP(arg1, ..., argn)'
```

At execution time, the number of arguments being passed into CUSTOMOP is compared to the list of symbols. If they match, then the program is ran with those values in the local variables. If they don't match, a "Wrong Argument Count" error is raised.

# →ARRY

**Member Of Menu:** Array
**Argument Types:** Real
Complex
List
**Result Type(s):** Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | | | |
|---|---|---|---|---|---|
| $Real_1$ | ... | $Real_n$ | $Real_{n+1}$ | → | [$Real_1$ ... $Real_n$] |
| $Real_1$ | ... | $Real_n$ | { $Real_{n+1}$ } | → | [$Real_1$ ... $Real_n$] |
| $Complex_1$ | ... | $Complex_n$ | $Real_{n+1}$ | → | [$Complex_1$ ... $Complex_n$] |
| $Complex_1$ | ... | $Complex_n$ | { $Real_{n+1}$ } | → | [$Complex_1$ ... $Complex_n$] |
| $Real_1$ | ... | $Real_n$ | { $Real_{n+1}$ $Real_{n+2}$ } | → | [[$Real_1$ ... $Real_n$]] |
| $Complex_1$ | ... | $Complex_n$ | { $Real_{n+1}$ $Real_{n+2}$ } | → | [[$Complex_1$ ... $Complex_n$]] |

This operation is used to create vectors and matrices from multiple values on the stack. If the top of the stack is a real value, the result will be a vector with that many values. If the top of the stack is a list with a single real value, the result will be a vector with that many values. If the top of the stack is a list with two real values, the result will be a matrix with that many rows and columns. The number of rows is the first value from the list.

The values for the vector or matrix are retrieved from the stack. All of those values must be real or complex. If any one of them is complex, then the result will be a complex vector or matrix. If all values are real, then the result will be a real vector or matrix.

# →HMS

**Member Of Menu:** Trig
**Argument Types:** Real
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This function takes a real argument which describes a time as hours and fractions of hours (the decimal component) and converts that time into hours, minutes, seconds and fractions of seconds. The result will be a number which looks like:

H.MMSS

The H value is the hour component. The two digits to the right of the decimal are the minutes component (MM) and the next to digits are the seconds component (SS) and further digits beyond those four to the right of the decimal are fractions of seconds.

# →LIST

**Member Of Menu:** List
Stack
**Argument Types:** Any
Real
**Result Type(s):** List
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

$Item_1$ ... $Item_n$ $Real_{n+1}$ → { $Item_1$ ... $Item_n$ }

This operation expects a real number at the top of the stack which is the size of the list it should create. Then, it pops off that number of items off of the stack and builds a list of those items. The items on the stack can be of any type, including other lists.

# →NUM

**Calculator Key:** ■→Num

**Member Of Menu:** None
**Argument Types:** Any
**Result Type(s):** Any
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Item_1$ | → | $Item_2$ |

This operation takes an item from the stack and evaluates it. For most types of stack items, evaluation does not change anything. For example, if 8 was on the stack before →NUM was executed, the result of →NUM is also 8.

However, for symbols, expressions and programs, the result may be different from the input. If a symbol is on the stack, then this operation will lookup that symbol, traversing directories from the current directory to the HOME directory. If it finds that symbol, it will replace it with the value of that symbol.

If an expression is on the stack, then any symbols in the expression will be evaluated as described above. The value of those symbols will be substituted in the expression. Any operations in the expression will be executed as long as the arguments for those expressions is known (for example a real value and not just a symbol). The final result may still be an expression or may be some other type, like a real, depending on the evaluation of the expression.

If a program is on the stack, then the program is executed. This may actually result in more values being popped off the stack or multiple values being pushed onto the stack. So, the stack diagram above may not be accurate depending on the program.

Note that unlike EVAL, →NUM will substitute the values of the global constants e, i and π in an expression.

# →STR

**Member Of Menu:** String
**Argument Types:** Any
**Result Type(s):** String
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | |
|---|---|---|
| $Item_1$ | → | $String_2$ |

This operation pops one argument from the stack and converts that item into a string. Whatever was displayed on the stack before is pushed back onto the stack as a string.

# ∂/∂x

**Calculator Key:** ▪∂/∂x

**Member Of Menu:** None
**Argument Types:** Symbol
Expression
**Result Type(s):** Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

$\text{Expression}_1 \ \text{Symbol}_2 \rightarrow \text{Expression}_3$

This operation determines the derivative of the expression against the provided symbol. Most operations in the calculator have known derivatives so expressions which are some combination of these built-in operations can be determined.

Derivatives can appear within an expression also but the syntax is a bit different. A derivative of SIN(X) against X looks like:

'∂X(SIN(X))'

So, the symbol to evaluate against appears after the derivative operation and the expression to evaluate appears inside brackets. When determining the derivative of a function off the stack by pressing the derivative button, the derivative is fully evaluated. That means that no "∂" operation will appear in the result. But, if you evaluate an expression with a "∂" operation in it, a single step in the derivative will be performed. So, in the above example, if you do an "EVAL" on that expression, the result will be:

'COS(X)*∂X(X)'

And then after another "EVAL":

'COS(X)*1'

For custom operations created by the user or for built-in operations with no known derivative, the calculator looks for a specific symbol which you can use to provide a derivative. For example, if you have created an operation called FOOBAR which takes two arguments, then you can try to evaluate an expression like this:

'∂X(FOOBAR(X,X))'

After evaluating this, you will get:

'derFOOBAR(X,X,∂X(X),∂X(X))'

The calculator prefixes the operation with "der". The "der" operation takes twice as many arguments as the non-"der" operation. In the case of an operation like FOOBAR which takes two arguments, derFOOBAR's first argument is the first argument to FOOBAR and the second is the second. The third argument to derFOOBAR is the derivative of the first argument and the fourth

argument is the derivative of the second.

As a way of an example, let's imagine that SIN() did not have a known derivative on the calculator (it does, but if it didn't...). We can provide the derivative by setting 'derSIN' to the following program:

« → x dx 'COS(x)*dx' »

This program encapsulates the proper derivative for SIN() and an application of the chain rule which says that the derivative of SIN(f(x)) with respect to x is COS(x) * f'(x). Using similar patterns, you can add derivatives for your custom operations.

NOTE: Because Halcyon Calc Lite does not include support for programs, you cannot define derivatives for custom operations on the free version of Halcyon Calc.

# Σ+

**Member Of Menu:** Stat
**Argument Types:** Real
Array
**Result Type(s):** None
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| $Real_1$ | → |
| $Array_1$ | → |

This operation takes a single real value or a single vector or matrix from the stack and adds the value(s) to the ∑DAT variable which holds the set of statistics values. The ∑DAT variable is always a real matrix which contains at least a single column of numbers. Each column represents a set of values for a single sample. The number of rows represents the number of samples in the statistics.

If the ∑DAT variable does not exist when this operation is executed, it will be created with the value(s) passed in. If a real value is passed in, the ∑DAT variable will be created and contain a 1x1 real matrix with the real value from the stack. If a real vector is on the stack, then the ∑DAT variable will contain real matrix with a single row. The number of columns in the real matrix will match the number of columns in the vector from the stack. Finally, if the argument on the stack is a real matrix, then that real matrix will be stored in the ∑DAT variable.

If the ∑DAT variable already exists, then the number of columns in the ∑DAT variable must match the number of values on the stack. If a real value is on the stack, then the ∑DAT has to be a real matrix with a single column. If it is, then the number of rows in the ∑DAT matrix is increased by one and the new value is added at the bottom of the ∑DAT matrix. If a real vector is on the stack, then the ∑DAT has to be a real matrix with the same number of columns as the vector from the stack. If so, then the ∑DAT variable has a new row added at the bottom with the values from the input vector. Finally, if a real matrix is on the stack, then the ∑DAT variable must have the same number of columns as the matrix from the stack. If so, then all rows from the input matrix are appended to the bottom of the ∑DAT matrix.

# Σ-

**Member Of Menu:** Stat
**Argument Types:** None
**Result Type(s):** Real
Array
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | |
|---|---|
| → | $Real_1$ |
| → | $Array_1$ |

This operation removes and returns the last sample from the statistics data. It expects to find a variable called ∑DAT which has a set of statistics values stored in a real matrix. Each row in the real matrix represents a single set of samples. Each column contains one of the set of values associated with a single sample. When ∑- is executed, the bottom row of values in the ∑DAT real matrix is removed and put onto the stack. If the ∑DAT matrix has a single column, the real value is pushed onto the stack. If the ∑DAT has two or more columns, then a real vector containing the values from the bottom of the ∑DAT matrix is pushed onto the stack.

After the operation finishes, the number of rows in the ∑DAT matrix is reduced by one. If there are no samples left in the ∑DAT variable, the ∑DAT variable is deleted.

# √

**Calculator Key:** ■√x

**Member Of Menu:** None
**Argument Types:** Real
Complex
Symbol
Expression
**Result Type(s):** Real
Complex
Expression
**Invertible:** Yes
**Valid In Expression:** Yes
**Stack Diagram:**

| | | |
|---|---|---|
| $Real_1$ | → | $Real_2$ |
| $Real_1$ | → | $Complex_2$ |
| $Complex_1$ | → | $Complex_2$ |
| $Symbol_1$ | → | $Expression_2$ |
| $Expression_1$ | → | $Expression_2$ |

This operation takes a real or complex value and finds its square root. The positive root is return for real values. If the input real value is negative, the result will be complex.

# ∫

**Calculator Key:** ▪∫

**Member Of Menu:** None
**Argument Types:** Real
List
Symbol
Expression
Program
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** No
**Stack Diagram:**

| | | | | | |
|---|---|---|---|---|---|
| $Expression_1$ | $Symbol_2$ | $Real_3$ | → | $Expression_4$ | |
| $Expression_1$ | $List_2$ | $Real_3$ | → | $Real_4$ | $Real_5$ |
| $Program_1$ | $List_2$ | $Real_3$ | → | $Real_4$ | $Real_5$ |

This operation can be used to determine the symbolic or numeric integral of the input expression or program. For symbolic integration, the arguments are the expression to integrate, a symbol to integrate against and a real number which should be a positive integer which is the degree approximation. Symbolic integrals are done by first determining a Taylor series approximation of the input expression (see the TAYLR operation for more information). The result of the Taylor series is a polynomial approximation which is then integrated using the simple rules for integrating polynomials. So, symbolic integrals are approximations unless the input expression is already a polynomial.

Numeric integrals can be determined for both expressions and programs. The second argument is a list which has information about the symbol against which to integrate and the bounds of the integral. The third argument is a real number which is the acceptable error in the result.

For example, to determine the integral of 'SQ(X)' from 0 to 1 against X to an accuracy of 0.00001, you would push these arguments onto the stack:

'SQ(X)'
{ X 0 1 }
0.00001

And then press the integral button to get the following results on the stack:

0.333333333333
9.87519399895E-06

The first value is the approximation of the numeric integral and the second number is an estimation of the actual error in that approximation and should be less than the error requested. Actual integration is done by evaluating the expression at "N" points between the two bounds to estimate the integral. The lower the error bounds you request, the more points are evaluated and

the longer the calculation will take. So consider how much actual accuracy you need because lower error bounds can significantly increase the execution time.

This same integral can be calculated using a program:

```
≪ X SQ ≫
{ X 0 1 }
0.00001
```

In this case, the integral is done with respect to X and X is used explicitly in the program. Finally, you can also do an integral of a program without specifying a symbol of integration. If the program pops a value off the stack and pushes a result onto the stack, then the symbol of integration is implicit. For example:

```
≪ SQ ≫
{ 0 1 }
0.00001
```

This program pops the value off the stack and pushes the square of that value onto the stack. The result of the integration with these arguments is the same as the previous two examples. The difference is that there is no explicit symbol of integration and the list argument has only an upper and lower bound.

# ≠

**Calculator Key:** ▪≠

**Member Of Menu:** None
**Argument Types:** Real
Complex
Integer
List
String
Symbol
Expression
Program
Array
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Real_1$ | $Complex_2$ | → | $Real_3$ |
| $Complex_1$ | $Real_2$ | → | $Real_3$ |
| $Complex_1$ | $Complex_2$ | → | $Real_3$ |
| $Integer_1$ | $Integer_2$ | → | $Real_3$ |
| $Real_1$ | $Integer_2$ | → | $Real_3$ |
| $Integer_1$ | $Real_2$ | → | $Real_3$ |
| $List_1$ | $List_2$ | → | $Real_3$ |
| $String_1$ | $String_2$ | → | $Real_3$ |
| $Array_1$ | $Array_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |
| $Program_1$ | $Program_2$ | → | $Real_3$ |

This operation produces a 1 if the two arguments are not equal, a 0 otherwise. It can operate on real, complex, integer, list or string values. It also can compare real and complex numbers and real and integer numbers. Note that a vector is equal if it has the same number of values and those values are the same. Similarly, a matrix is equal if it has the same number of rows and columns and each value is the same.

# ≤

**Calculator Key:** ▪≤

**Member Of Menu:** None
**Argument Types:** Real
Integer
String
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Integer_1$ | $Integer_2$ | → | $Real_3$ |
| $String_1$ | $String_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation takes two real, integer or string values and produces a 1 if the first value is less than or equal to the second, 0 otherwise. The result is always a real, even if the incoming arguments are integers or strings.

# ≥

**Calculator Key:** ▪≥

**Member Of Menu:** None
**Argument Types:** Real
Integer
String
Symbol
Expression
**Result Type(s):** Real
Expression
**Invertible:** No
**Valid In Expression:** Yes
**Stack Diagram:**

| | | | |
|---|---|---|---|
| $Real_1$ | $Real_2$ | → | $Real_3$ |
| $Integer_1$ | $Integer_2$ | → | $Real_3$ |
| $String_1$ | $String_2$ | → | $Real_3$ |
| $Symbol_1$ | $Symbol_2$ | → | $Expression_3$ |
| $Expression_1$ | $Expression_2$ | → | $Expression_3$ |

This operation takes two real, integer or string values and produces a 1 if the first value is greater than or equal to the second, 0 otherwise. The result is always a real, even if the incoming arguments are integers or strings.