

IntCalc

version 1.1

An Integer-oriented RPN calculator (but with floating point too).

Brought to you by:

The Office of Dubious Programming

Copyright © James Preston, 1992

IntCalc is shareware. If you like it, please send \$10 to:

James Preston

1904 Miraplaza Ct. #23

Santa Clara, CA 95051

Bug reports and enhancement requests are encouraged.

IntCalc was developed using Symantec's THINK C.

What's new for version 1.1

- Due to popular demand, *IntCalc* now comes in a Desk Accessory version. DAs are a pain to write, and they're unnecessary in system 7.0, but enough of my paying customers requested it, so who am I to argue? If you're happy with the application version, feel free to ignore or trash the DA, you won't hurt my feelings. If you do want to use it, please read the **What you should know about the DA version** section below first.
- Despite almost no demand (one customer), *IntCalc* is now programmable. I had wanted to do it anyway, and it was kind of fun. I hope at least a few people make use of it. See **The Power of Programming** below for full details.
- The floating point mode now uses extended format (10-byte SANE, for those who know or care). This allows for 16 decimal places of precision. One consequence, however, is that integers are no longer the same physical size as floating point numbers, so the option of switching modes while leaving the bit pattern intact is no longer provided. I hope no one ever used that. Another consequence of this is that it no longer makes any sense to allow the bitwise AND, OR, XOR, and NOT in floating point mode. However, the RMD (remainder) now does a real floating point remainder.
- For those with color Macs, the buttons are now color-coded in groups according to some rough idea of functionality (e.g. all integer-only buttons are one color, all floating-point only buttons are another color). I tried to use pleasant (or at least non-gross) colors. I didn't have time to provide user-definable colors, but I did add an item under the **Edit** menu that lets you go back to plain black & white buttons if you really don't like my color choices. Also, the colors are stored in a 'cctb' resource, so if you know how to use ResEdit, you can change the colors that way.
- An alternate format, using comma (",") as the radix character and period (".") as the separator is now available (see the discussion of the SEP button below).
- "IntCalc data" will now be placed in the "Preferences" folder if it exists.
- *IntCalc* should now work properly on systems with more than one monitor (that is, you can drag the windows anywhere on the available real estate, rather than being restricted to the main monitor as in version 1.0).

If you're reading the MSWord version of this document, version 1.1 changes are marked with change bars like what you see on the left; users of version 1.0 can read just the new sections. If you're reading the MacWrite version, ignore this paragraph.

Oh no, not another calculator tool!

Yes, but this one really does have some features I haven't seen in other tools.

One problem I have with the calculators-on-computer I have seen is that they try to duplicate a calculator exactly, including all its limitations. Why put a calculator on a computer if you don't make use of the added capabilities?

The *IntCalc* display shows you all four elements of the stack all the time, and is wide enough for all 32 bits of a binary number.

It has an optional window that provides a second view of the stack with independent control of each number's base, so you can see a number in both decimal and hex (or any other two bases) at the same time.

It has another optional window that displays the contents of the 16 storage registers, so you can see what's where.

And, in addition to the expected binary, octal, decimal, and hexadecimal displays, *IntCalc* can also display in ASCII.

(On the other hand, to show that I am not immune from the "thinking too much like a calculator" syndrome, it was brought to my attention that *IntCalc* would be more Mac-like if it allowed full editing of the stack and register fields. Maybe in the next version.)

Oh, and despite the name, *IntCalc* does have floating point.

What you should know about the DA version

The first thing you might notice is that some of the buttons don't look the same. In the application version, I use a custom font to get the down arrow on the roll-down key, the double arrow in the x-y exchange key, and some others. Unfortunately, fonts cannot be attached to a DA, so the DA version cannot use my custom font. I did the best I could with the standard characters, but it's still not the same. Another consequence of this is that, for the DA version to look right, you need to have the Geneva font in size 10 installed in your system.

The next thing you might notice is that the auxiliary windows (alternate view, etc.) don't have close boxes. The problem is, I guess, that DAs were not designed to have multiple windows. So when the system sees a click in the close box of *any* DA window, it just sends the DA a message to go away. I was unable to find a way to detect which window's close box really received the click. So you'll have to use the calculator's buttons to dismiss the windows.

Nifty on-line help (or, what's an LDZ button?)

For those who don't read documentation, or those who just need a little memory boost once in a while, *IntCalc* provides a little direct help. Hold down the command key and click and hold on a button. A little window will popup with a short description of what that button does. (I know it's not exactly System 7 balloon help, but I did think of it before hearing about System 7, and I like the instant-access better than having to click in the menu bar before and after. I know I probably should have added balloon help too, but version 1.1 is already almost a year late, and balloon help is not that easy to do.)

Where the heck is the "=" button on this thing?

For those not familiar with the HP-style Reverse Polish Notation (RPN), I think your best bet would be to find a friend who has an HP calculator and get a lesson, or at least borrow the instruction book. Oh, heck, I'd better say a couple of words about it here, otherwise I can't really call this "documentation".

Briefly, the working part of the calculator consists of a four element stack. The elements are labeled X, Y, Z, and T (don't ask me about that last one, I just use HP's names). When you type digits, the number goes into X. When you are done typing a number, click the Enter button. This "pushes" the stack: The contents of X move into Y, the contents of Y move into Z, the contents of Z move into T, and the contents of T go into the bit bucket. Temporarily, X still contains the entered number. If the next button you click is a digit, it will overwrite what is

displayed in X. If, on the other hand, the next button you click is an operation, it will operate on what you see in X.

Clicking on a binary operator (addition, subtraction, etc.) performs that operation on X and Y in the order Y <op> X. So, for example, to compute 5-3, you would do the following:

- Click (or type) 5
- Click Enter (or type **return** or **enter**)
- Click (or type) 3
- Click (or type) -

At this point, the stack drops (the value in Z moves into Y; the value in T is copied into Z; T remains unchanged) and the result (in this case 2) is put into X. The former value in X (in this case 3) is copied into a special register called "last X". This value can be recalled by clicking on the LSX button.

For unary operations, such as SHL, the result replaces the previous value of X and the rest of the stack is unchanged. And again, the previous value of X is copied into the "last X" register.

For those who'd rather type than click

As expected, you can enter digits by typing. You can also enter ASCII characters this way, when the base is ASC. The **delete** key acts just like the BSP button (or vice versa, depending on your point of view), and the **return** and **enter** keys act just like the Enter button. In FLT mode, the **e** key acts like the EEX button.

If you have an extended keyboard, the "/", "*", "-", and "+" keys on the numeric keypad will perform the associated operation. Note, however, that the same keys on the regular keyboard do not perform the operation but instead type the associated character (this is so that you can enter those characters in ASC mode). The **clear** key on the numeric keypad acts like the CLR button.

The command keys for **undo**, **cut**, **copy**, and **paste** are also enabled in *IntCalc* (see below for more information on how cutting and pasting works in *IntCalc*).

From binary to ASCII to floating point

The five bases BIN (binary), OCT (octal), DEC (decimal), HEX (hexadecimal), and ASC (ASCII) are collectively referred to as integer modes. Clicking on one of these buttons converts all elements of the stack into the corresponding base (ok, for you sticklers, it doesn't actually convert anything it just changes the base used to interpret the bit pattern). You may also individually set each stack element to any of these bases via the popup menus on the right.

FLT (floating point) mode is completely separate and may not be intermixed in the stack with the integer modes. The popup menus are disabled when in FLT mode. When you click on the FLT button, you will be prompted to click on a digit button to specify the number of digits to display to the right of the decimal point. Note that, from within FLT mode, you can change the number of displayed decimal places by again clicking the FLT button followed by the new number of places to display. This affects the display only; internally, the values are always stored in full precision.

When *IntCalc* is changed from an integer mode to FLT mode or vice versa, the stack is not cleared and the values therein are converted (as best they can be) from the old base to the new. Thus, if X contains a decimal 5, when you switch to FLT mode X will contain a floating point 5. The values are truncated when going from FLT to an integer mode, so if X contains floating point 5.6, switching to decimal will give you an integer 5.

IntCalc v1.1 uses the Macintosh's SANE extended precision floating point numbers, which are 10 bytes wide. The integer values are only 4 bytes wide. This means that when the main calculator is in FLT mode, any values shown in the alternate view window (which is always in integer mode) are not meaningful, since they only show a part of the corresponding floating point numbers.

Certain functions are valid in only one mode or the other. These buttons are disabled (which makes them dimmed and mostly unreadable) when they are not valid. In integer mode, the "."

(decimal point), " \sqrt{x} ", " $1/x$ ", " y^x ", and EEX buttons are dimmed. In FLT mode, the SHL, SHR, ASR, SLn, SRn, ARn, RL, RR, SB, RLn, RRn, CB, MSL, MSR, #B, XOR, AND, NOT, and OR buttons are dimmed. The LDZ button is not disabled in FLT mode because you might want it for the alternate view window. The RMD button in FLT mode does a floating point remainder.

To comma or not to comma

To aid readability, *IntCalc* puts a comma between every three digits in DEC and FLT modes, and between every four digits in BIN and HEX modes. If you'd rather see all the digits scrunched together, the SEP (separators) button will toggle this display off and on.

The usage of the period for the radix mark and comma for the separator can be reversed by holding down the **option** key while clicking on the SEP button. This allows *IntCalc* to conform to the numerical convention in many countries.

If you prefer to always see all the digits, the LDZ button toggles the display of leading zeros. This applies only to BIN, OCT, and HEX modes.

Squirreling things away for the future

IntCalc has sixteen storage registers. To store the value from X into a register, click the STO button followed by one of the digit buttons (0 thru F). The value in X will be copied into the designated storage register. To recall a value, click the RCL button followed by the digit button of the desired register. The stack will be pushed, and the stored value will be entered into X.

The RGS button brings up a window showing the contents of the sixteen storage registers. Each will be displayed in whatever the base of X was when it was stored. The CLEAR ALL button in this window will reset all storage registers to zero.

Cutting and Pasting

Like all good Macintosh programs *IntCalc* allows cutting and pasting, via both the Edit menu and the command-key equivalents.

Undo (not too surprisingly) reverses the last change to the stack.

Clear resets X to zero.

Cut and **Copy** copy X to the clipboard, with **Cut** having the added effect of clearing X.

Paste copies the clipboard to X, pushing the stack first (even if the calculator was in digit entry). *IntCalc* looks at the characters to be pasted and tries to handle the pasting intelligently using the following rules:

If *IntCalc* is in floating point mode then

It will attempt to interpret the value to be pasted as a floating point value. If it is not a valid floating point value, nothing will be pasted.

If *IntCalc* is in an integer mode then

If the value to be pasted starts with "0x" then

If what follows the "0x" is a valid HEX value then

the base will be changed to HEX and what follows the "0x" will be pasted.

(This is standard C notation for indicating hex values. e.g. if the clipboard contains "0xA34F" then the hexadecimal value "A34F" will be entered into X.)

else

the base will be changed to ASC and the first four characters of the value will be pasted (including the "0x"). (e.g. if the clipboard contains "0xBACK" then "0xBA" will be entered into X in ASC mode.)

If the value to be pasted contains all digits then

If the value is valid in the current base then

the value will be pasted in the current base.

else

the base will be changed to DEC and the value will be pasted.

else if the non-digit characters are valid HEX digits then

the base will be changed to HEX and the value will be pasted.

else

the base will be changed to ASC and the first four characters of the value will be pasted.

If the above seems a little confusing, just forget it. Most of the time when you paste something, you'll know what you're doing and it will work the way you expect it. Only when something unexpected happens should you need to refer to the above to find out what happened.

For those who occasionally make mistakes

When an invalid operation is attempted, such as division by zero or attempting to make a mask bigger than the word size, *IntCalc* beeps and displays a little window with an error message in it telling you what went wrong. The next time you click the mouse or type a key, this window will go away.

A change from version 1.0: Clicking on a button (or typing a key) while the error window is displayed will no longer perform the action of that button or key.

If you know what the "C" stands for in "16C", raise your hand

IntCalc duplicates the continuous memory feature by creating a file in the system folder called "IntCalc data". The following information is read from this file when you start *IntCalc* and stored into the file when you quit:

The contents and base of each stack element.

The base of each stack element in the alternate view window.

The contents and base of all storage registers.

The contents of the lastx register.

The state of the leading zeros (LDZ button) and show separators (SEP button) flags.

The number of displayed digits in FLT mode.

The position on the screen of the calculator window, the registers window, and the alternate view window.

The visibility state of the registers window and alternate view window.

The position and size of the programming window.

The program.

Whether the buttons are displayed in color on a color monitor.

You can also use the **Save** and **Open** menu items to create and restore your own data files, and, of course, you can double click on a data file from the finder to start *IntCalc* with that data.

IntCalc v1.1 knows about the **Preferences** folder. When starting up, *IntCalc* looks first for "IntCalc data" in the **Preferences** folder. If either the data file or the folder don't exist, then *IntCalc* looks in the system folder proper as before. When you quit, the data file will be written into the **Preferences** folder if the folder already exists (*IntCalc* will never create the folder). Additionally, if the data file used to be in the system folder, that file will be deleted.

Don't tell anyone, but I violated the interface standards

I've never liked the edict that the first click in an inactive window just activates it. Why should I have to click twice to use a button in another window? This is especially bothersome in a multi-window application like *IntCalc* wherein the concept of a single active window doesn't really apply. The main calculator window is where all the action is, and using the button or pop-ups in the other windows doesn't change that. So, my violation is two-fold: No matter which window is active, typing always goes to the *IntCalc* window. And clicking once on a button or pop-up in an inactive window not only makes that window active but also operates the button or pop-up. Note, however, that this last only applies among *IntCalc*'s own windows. If another application or DA is active, then a click on *IntCalc* will only activate it, not operate a button.

The Power of Programming

"Programming" *IntCalc* is simply a matter of recording a sequence of keystrokes. Almost all buttons can be recorded in a program; the exceptions are BSP, RUN, CONT, SST, "?", "Redisplay", and "Clear Pgm". In addition to the usual calculator functions, there are several programming-specific buttons for handling branching, looping, and subroutines. These are available on the programming window.

Creating a program. On the right side of the calculator body is a switch. The top part of the switch indicates the current mode, which will be "calc" when you first run *IntCalc*. Click on the switch to enter programming mode. The indicator will now read "prog" and the programming window will automatically come up. In this mode, any button clicked or any key typed (with the exceptions noted above) is not executed, but rather stored into the program at the current insertion point.

Executing a program. Click on the RUN button, and your program will start executing from the beginning. You can do this from either "calc" or "prog" mode. The X line on the calculator body will display "running". When the program finishes, *IntCalc* will beep, and all the displays will be updated. (I experimented with having it update the displays as the program executed, rather than just displaying "running", but it made the programs run about three to five times slower.)

If you want to start executing from a particular label, hold down the **option** key and click RUN. You will be prompted for a label. Click the button corresponding to the desired label, and the program will go there and start executing. This enables you to have a number of independent sub-programs in the same program; each can be accessed via its label.

Why a program stops. If a program encounters an error that prevents it from proceeding, such as an illegal digit, or an attempted branch to a non-existent label, it will stop and put up the error window telling you what went wrong.

If you want to deliberately stop a running program, perhaps because you think it might be in an infinite loop, just type a key or click the mouse and the program will stop where it is.

Otherwise, when the program ends normally, *IntCalc* will beep and all the displays will be updated.

The program counter (PC). In the programming window, the program counter (PC) will show up as a ">". The PC points to the instruction that will be executed next. The PC indicator will walk through a LBL or a GTO; that is, first you will see ">GTO 5", then a click on the SST button (see below) will show "GTO>5". However, it will not walk through numbers. If you see ">12345", clicking once on SST will still show ">12345", even though the "1" has executed and will show up in X. I emphasize that it is only the *indicator* that doesn't move; the program counter itself still moves forward.

Starting up from where you are. Clicking on the CONT (continue) button will continue program execution from the current PC.

Clicking on the SST (single-step) button will execute just the instruction at the PC, and then return control to you.

Editing a program. Use the mouse to position the insertion point anywhere in the program. Use the BSP button or the **delete** key to remove the instruction to the left of the insertion point. Click other buttons to insert instructions at the insertion point. You'll undoubtedly soon notice that if you click in the middle of a button-mnemonic, the insertion point jumps the beginning of that mnemonic. You'll also notice that if you try to select something, *IntCalc* stubbornly ignores your selection and reverts to a single insertion point.

Why you can't select anything in the programming window. This is likely to be a real sore point with Mac purists, and I apologize for any frustration that it causes. The problem is that the text shown in the programming window is not *really* text in the usual Macintosh sense. Instead of thinking of each character as a discrete, editable unit, you should think of each button-mnemonic as a discrete unit.

Internally, the program is stored as a sequence of byte-size codes, one for each button. I write the text to the programming window on the fly; the text itself is never actually looked at

by the internals of *IntCalc*. If I allowed you to delete any arbitrary sequence of characters, or insert into any arbitrary location, I would then have to implement a full-blown parser to determine the meaning of the resultant text. I'm sure that anyone familiar with writing compilers can appreciate why I didn't really want to do that for this little calculator.

(So why didn't I just disable the ability to make selections? I wish I could. But there is a single Macintosh toolbox call that enables the positioning of the insertion point and also the making of selections. I can't get one without the other unless I write my own replacement for the toolbox routine, and that's a little beyond me.)

Does your program suddenly look funny? I tried to make the program display logical and convenient; so operators (like "+" or XOR) show up on a line by themselves; things followed by a label (like LBL or GTO) include the label on the same line; the conditionals put the following instruction on the same line, to reinforce the interpretation; and any sequence of digits gets put on one line, since it should be one number. But occasionally, with various insertions and deletions, my internals lose track of my externals, and the program display puts a carriage return where it doesn't belong, or forgets to put one where it does. So if you ever see two instructions on one line or anything else that doesn't look right, just click the Redisplay button and the entire display will be redone from the internal program. Let me hastily add that this *should* never happen; I did a lot of testing, and tried to account for all possibilities. But experience has shown that there is usually at least one user who will try something that the author never dreamed of. So the Redisplay button is provided just in case.

Getting around in the program. Simple unconditional branching is done via the GTO *label* instruction. The *label* can be any single digit (0 thru 9, A thru F, or I [see below]). The branch will go to the LBL *label* instruction with the matching *label* (it is not possible to branch to an arbitrary instruction). If more than one LBL exists with the same *label*, only the first one (closest to the beginning of the program) is accessible.

GSB *label* also branches to the corresponding LBL instruction, however execution automatically transfers back to the instruction following the GSB when an RTN (return) instruction is encountered.

The mysterious "I" button. It's for the index register. In "calc" mode, you can use it as just another storage register. In "prog" mode, it's used for indirect branching. The instruction "GTO I" will transfer control to the label that corresponds to the value stored in the index register. So, for example, if the index register contains 7, "GTO I" will branch to "LBL 7". Similarly, "GSB I" can be used to indirectly call a subroutine.

If the value in the index register is in floating point, only the integer portion is used. So, for example, if it contains 1.5, the branch will go to LBL 1. If the value in the index register does not correspond to any existing label, the program will stop with an error.

Conditional tests. The second and third columns of buttons on the programming window provide you with eight different tests for various combinations of comparing X and Y or comparing X and zero. These follow the "Do if true" rule: execution proceeds to the next instruction if the condition is true; execution *skips* the next instruction if the condition is false. Usually, the most useful instruction to place after a condition is a GTO, but it can be anything you want.

In the third column are two conditionals that work slightly differently. The DSZ (decrement and skip if zero) and ISZ (increment and skip if zero) instructions are usually used to control a loop. For example, you store into the index register the number of times you want the loop to execute. At the bottom of the loop, use the DSZ followed by a GTO to the top of the loop. The DSZ will subtract one from the value in the index register and store that new value back into the index register. If that value is equal to zero, the next instruction (the GTO) will be skipped; otherwise it will be executed and the program will go through the loop again.

How the program gets data. You can write the program to get its data from the storage registers, in which case you need to put the appropriate data into the appropriate registers before starting the program. You can also write the program to get its data from the stack. In that case, you can either enter the data into the appropriate stack elements before starting the

program, or you can use the PSE (pause) button to interrupt the program at various places and enter the data as needed, then click the CONT button to continue.

Saving and getting programs. Programs are saved into *IntCalc* data files along with everything else. If you have only one program, you need do nothing special; the program will be saved into "IntCalc data" and will be there the next time you run *IntCalc*. You can also use the **Save** command to store specific programs, and retrieve them with the **Open** command.

Cautions and limitations. No checking is done while you are entering a program to see if it will actually be legal when executed. This means, for example, that you can freely enter a sequence like "DEC 12AB". When you run the program, however, and the PC gets to the "A", the program will stop and the error window will popup informing you of the invalid digit. (Note that this behavior is slightly different than when you type an invalid digit normally; when a program is running, you don't get the first warning beep.)

The total size of an *IntCalc* program is limited only by the memory available in your Mac. However, you are limited to only sixteen labels, so that might put a damper on writing really huge programs (and if you're writing something really huge, why are you doing it on a dinky little calculator anyway?)

Version history

October 25, 1990 Release of version 1.0

June 1992 Version 1.1

Bugs fixed:

- 1) If you stored something into memory in an integer mode, then restored it from FLT mode, the value was not being converted.

Enhancements added:

- 1) DA version.
- 2) Programmability.
- 3) Floating point is now in extended format.
- 4) "IntCalc data" goes into "Preferences" folder if it exists.
- 5) Optional alternate radix format.
- 6) Color.
- 7) Multiple monitor support.