[9.2] User interface considerations

These suggestions will help make your programs easier to use, and less frustrating. The goal of any program is to enable the user to accomplish a task as quickly and efficiently as possible, and I think these ideas work towards that goal. In almost all cases these suggestions will make your program larger, but rarely make it slower. Not all suggestions are appropriate for all programs.

- Check user inputs for data type and range. If the input should be an integer, warn the user if he enters a string or floating point number, and give him the chance to try again. If the input should be between certain limits, and he enters a number outside those limits, display a warning and let the user try again.
- If a program needs lots of inputs from the user, save those as global variables so they can be used as defaults with Request in dialog boxes.
- Trap errors (with *Try ... EndTry*), and display useful warning messages. The message should display both *why* there is a problem, and *what* to do to fix it. Many errors can be trapped before the code even runs: think about what the program will do with various error conditions, or different values for variables.
- Use the Title statement in dialog boxes to help the user. For example, you can use titles of INPUT, RESULTS, ERROR and WARNING to make the message purpose more clear. I find that all caps is easier to read in the small font used in the dialog box title.
- Don't change the mode settings without notifying the user. The Mode settings include the angle mode (radians, degrees), display format, complex number format, graph type and so on. If you must change the modes, notice that *exact(), approx(),* ˜, and ¬ can be used to temporarily over-ride the arithmetic mode and angle mode. If you have to change other modes, use *GetMode("ALL")* and *SetMode()* to save and restore the user's mode settings.
- For large, complex programs consider including a 'help' menu item or custom key. The help should explain how to use the program and any special features or limitations. The help should be built into the program so that the user can see it while running the program. Ideally, the help code and text strings should be very modular, and your program should provide an option to delete the help system, since this uses up RAM very quickly. In this way, inexperienced users get the help they need to run your program, and they can remove it when they don't need it anymore.
- Set the application's folder within the program itself. This makes sure that all subprograms, functions and variables are available while the program is running. When the program exits, restore the folder so that the user doesn't have to, and isn't surprised to be in a different folder.
- Use local variables if at all possible. However, you will be forced to use global variables in some cases. Use DelVar to delete these automatically when the program exits. If you think that the user might want to keep them, give her the option to delete them when the program exits.
- If your program uses matrices or other data structures, it is most likely that those are not in the application's folder. So, when you prompt for the variable name, make sure that the user knows that he needs to enter the folder name, as well. Sometimes I prefer two separate Request statements, one for the folder name, and one for the variable name. Your program needs to be able to refer to variables by both the folder name and the variable name.
- If your program creates large data structures, check available memory with *getConfg()* before creating the variable. If there isn't much free RAM, warn the user.
- If your program uses large data structures, think about archiving them automatically, so they don't use so much of the calculator's RAM. If your program only needs to read from the data structure, it can remain permanently archived. However, if the program needs to write to the data structure, you will need to unarchive it, perform the write operation, then archive it again. If you try this approach, make sure that your program cannot execute a loop in which the variable is rapidly and repeatedly archived. In extreme cases this will 'wear out' the archive flash memory.

- Provide an 'exit' or 'quit' menu item in your program. This gives the program a chance to delete global variables, restore the mode settings, and restore the folder.
- If your program displays output on the program I/O screen, use *DispHome* when the program exits. This means the user doesn't have to press [green diamond] HOME to get back to the home screen.
- Consider providing a menu to let the user set the number display digits and exponential mode, for numeric results. Use *format()* to display the results based on the user's choice. In large programs with lots of output, I even let the user set different formats for different sets of variables.
- If possible, use functions and commands that are available on both the original 92 as well as the 89/92+. This means that more people can use your program. However, this is a common dilemma encountered whenever hardware and software improves. In some cases, this means that users with the newer systems pay a penalty in terms of execution speed or code size.
- Keep in mind that there are two broad types of users. The first type just wants an answer from your program from the command line, as quickly and easily as possible. This user needs dialog boxes to prompt for the inputs, and clearly labeled output. The second type of user might want to use your code in her own programs. This user wants your code as a function, not a program, so it can be easily called, and return its results to the user's program. One clean way around these conflicting requirements is to encapsulate your core functionality as a function, and supply an additional interface routine that provides prompts and labelled output.
- Provide documentation for your program. You might have written the greatest program in the history of coding, but if people can't figure out how it works, you might as well have saved yourself the effort. The documentation doesn't need to be a literary work of art.

In the end, it is your program, and you'll write it the way you want to. And writing a robust, efficient program is a lot of work. It is likely that if you use all of these suggestions, the amount of code needed to make the program a pleasure to use will be far greater than the code to actually do the work! Depending on the program, it might not be worth the effort. You just have to use your judgement.