[7.32] Documenting programs

Program documentation answers these questions about your programs:

1. What problem does the program solve?
2. How do I use the program?
3. How does the program get its results?
4. How can I change the program to better meet my needs?

Documentation is work and it takes time. For most programmers, documentation is a chore, and not nearly as interesting as designing and coding. If you are just writing programs for yourself, the documentation is obviously your own business. However, if you distribute your programs to others, I assume you really want people to be able to use them. Some useful programs may be so simple that they don't need any documentation, but I have seen any, yet.

The main goal of documentation is to answer the questions above, as quickly as possible for both you and the user. While you may be rightfully quite proud of your program, the user probably does not share your enthusiasm. The user needs to solve a problem, and it looks like your program might do it. Your responsibility in creating the documentation is to meet the user's needs. Those needs can be grouped in two broad categories: answering questions and giving warnings. Some of the questions are obvious, such as: how do I install the program? how do I run it? how do I fix this error? The warnings are less obvious but just as important. You need to inform the user as to side effects that might not be obvious. Some examples of these side effects might be changing the modes, creating folders or creating global variables.

Documentation is just writing. Clear writing is a skill which, like any other, takes practice and commitment. The more you do it, the better (and faster) you will get. In the remainder of this tip, I give some suggestions for creating usable documentation. The suggestions are organized around the four questions that users have.

What problem does the program solve?

If you have written a program to solve a problem, you are quite familiar with the problem. It may be a problem in your field of expertise that you solve every day. But other users don't necessarily have your background. They might just be learning about the field. They might be unsure of the terminology and solution techniques. To meet those users' needs, you must have a short, clear, complete description of the problem.

How do I use the program?

Your documentation should answer these questions for the user:

- Does the program run on either the 89 or the 92+? Are there special versions for each calculator?
- Does the program require a specific version of the AMS?
- How much ROM does the program take?
- Can the program be archived?
- How do I install the program?
- How do I uninstall the program?
- Does the program need other functions or programs?
- Can the program be intstalled in any folder, or does it have to be in a specific folder?

- How do I start the program?
- How do I exit (quit) the program?
- What Mode settings are needed? Does the program change my Mode settings? Does the program restore my mode settings?
- Does the program change the current folder? Does it restore the previous folder?
- Does the program change my Graph screen? Does it change my Program I/O screen? Does the program change any of my Y= Editor settings?
- Does the program leave any Data plot definitions behind?
- Does the program leave behind any global variables? What are there names? What happens if I delete them, then run the program again?
- What other side effects does this program have?
- How does the program get its inputs from me? What are valid inputs? What are the units for the inputs? Can the inputs be real or complex? If my input is a function name, how do I specify that function?
- If this is a function, how do I call it? What are the calling arguments? Give examples.
- How does the program display the output results? What are the units? Are the results stored anywhere?
- How accurate are calculated results? How can I check the accuracy? Does the accuracy degrade for some input ranges?
- How long does it typically take for the program to get an answer?
- If the program duplicates a built-in function, why is it better? Is it faster, more accurate, or is it more general?
- How much testing did you do on this program?
- Does the program display any error messages? What do the error messages mean? How do I fix the errors?
- Does the program have built-in help? How do I get to it? Can I delete it to save memory?
- Where do I get help, if I need it?
- What is the version of this program?
- If this is a new version of the program, how is it different from the previous version?
- Does this program have any known bugs? How do I avoid them?
- Is this program based on someone else's work? How did you improve on their effort?

This is a fairly complete list, and all of these items might not be appropriate for all programs. You just have to use your judgement as to what best serves the user.

How does the program get its results?

This section of the documentation describes *how* the program does what it does. For various reasons, you might not want to tell the user how your program works. I believe this is a disservice to the user. If I am counting on your program to give me a correct answer, I deserve some assurance that it really works. In part, this assurance can come from knowing what algorithms you used to get the results. If the calculation depends on some built-in function, for example, *solve()* or *nint(),* and you tell me that, then I know that your program is limited by the performance of those functions, and I can evaluate the results accordingly.

This section of the documentation can be fairly short. If I really want to know all the intricate details, I can examine your source code. The real purpose of this section is to give an overview.

How can I change the program to better meet my needs?

It may be quite likely that your program almost does what the user wants. In this case, the user might want to modify the program. To do this, she needs to understand how the program works, and this means code documentation. Code documentation differs slightly from user documentation, because both the intended audience and the purpose are different. User documentation meets the user's needs, and the user is not necessarily a programmer. Code documentation meets the needs of a programmer. Useful code documentation is actually a field of its own, and I'm only going to give a brief overview of 'best practices', based on my personal experience of programming and supervising programmers. The documentation also interacts to a great degree with the underlying design of the code, which is a topic that deserves a tip of its own.

A program consists essentially of data, calculations, and flow control. Because these three attributes interact with each other, the documentation must specify the interaction, as well as the characteristics of each of attribute. Data documentation consists of

- Choosing a variable name. Ideally, variable names are long and descriptive. On the 89/92+, memory is limited, so there is a strong motivation to keep variable names short, to save memory. In any case, variable names are limited to eight characters, which often precludes truly descriptive names. This means that descriptive documentation of variables is even more important. Some simple conventions can help, as well. For example, integer loop counters are traditionally named with integers *i, j, k, l, m* and *n.* Nested loops can use loop counters such as *k1* and *k2*. Some 89/92+ programmers prefer to use Greek or non-english characters, especially for local variables. Selecting variable names on the 89/92+ is further confounded by the fact that the AMS uses several common, obvious names as system variables and data.
- Describing the data type: is it a variable or a constant? Is it an integer, a string or a matrix? What are the valid ranges and dimensions? What is the variable's *physical* significance? For example, *Vin1* and *Vin2* may be two circuit input voltages, and *Vout* is the output voltage.
- Clarifying variables which change type during the program execution. It is completely legal in TI Basic to store a string to a variable which contains an integer, for example. While this improves efficiency, it can make programs very difficult to understand. Further, it is common (and efficient) to 'reuse' a local variable when the previous contents are no longer needed.

Documenting calculations is fairly straightforward, particularly if the underlying data is well documented. Keep in mind, though, that you want to describe the functional intent of the calculation, not the calculation itself. For example, this isn't too helpful:

```
©Add 273.15 to temp1
temp1+273.15→temp1
```

This is better:

```
©Convert temp1 from degrees C to degrees K
temp1+273.15→temp1
```

Some calculations can be efficiently coded as a single large, complex expression. These are the calculations that really need good explanations. Remember that the purpose of the description is to describe *why* you are doing something, not *what* you are doing - the 'what' is obvious from the code itself.

Documenting the flow control is easy if the program structure is logical to begin with. *goto*'s and labels are discouraged in good structured programming, for good reason, but they can result in the most possible efficient code when used sparingly. It can be challenging to come up with good label names, so I just have a few common names for pervasive elements such as *mainloop* and *exit*. Error handling may interrupt normal program flow, so the program comments should make it clear *where* the error is handled, and *where* control is transfered after the error is handled.

Where to put the documentation?

There are at least three possiblities for physically locating the program documentation: in the source code, as a separate *readme* text file, or as an 89/92+ text file. Putting the documentation in the source code file has the advantages that the documentation always travels with the program, but it dramatically increases the code size and uses up valuable on-calculator memory. Programs are most commonly documented with a separate *readme* file. This certainly works, but then the user might not have access to the documentation when they really need it most.

Creating an 89/92+ text file that includes all the program documentation has these advantages:

- The documentation is on the calculator, with the program, and can be readily accessed. If one user transfers the program to another user, by calculator, he can also send the text variable, so the new user has the documentation, too.
- You can use the special 89/92+ characters in the documentation, which is not easily done in a separate *readme* file.
- You can include sample calls in the text file, which can be executed directly from the file.
- The user can edit the text file to meet their needs. For example, when they first start using the program, they may need most of the detailed documentation. As they become more familiar with the program, they can delete the text they don't need to save memory. Finally, they can add their own particular comments as needed.

If you use this method, you should give the name of the documentation text variable in the source code.