WINDOWS ASPECT: A Scripting Language

A Tutorial - Part Two for Procomm for Windows v.2
GHOST BBS v.3.20 © 1995 by *Gregg Hommel*

When last we left our intrepid hero, George, he was just starting to learn about scripting in the Windows Aspect (Wasp) language of Procomm Plus for Windows 2.0, by writing a *very* simple log on script.

Let's give him a little time to get in some practice, before we pick up his saga and look at something that we touched on briefly in the last column, i.e. the nature of a script. What we said way back then was this....

“In Wasp, a script is a series of commands which Procomm Plus for Windows will read when told to, and execute as specified. These commands are written using a particular form, following a designated syntax, and instruct Procomm Plus for Windows to perform various tasks at specified times, and in a specified order."

Now, to some of you, that just might look like a rather simplified definition of a computer program. This could be because a script can be considered just that, and those of us who write scripts generally consider what we do as programming.

Basically, I suppose that every programmer has his own idea of what makes up good programming practice. In my opinion, this good practice consists of two basic routines :

1) planning in advance what you want the script to accomplish, and roughly how it might do so, and

2) writing the code you need in a modular fashion. Let me explain.....

It is quite difficult to write code to have a script do what needs be done next, when you, the author, have no idea what you want it to do next. The code won't "get anywhere" if you, the programmer, have no idea where you expect it to go.

Generally, before I begin work on any code, I attempt to write out, in English, what I want the code to accomplish, and where I want it to finish when done. This gives me a basic word picture of what I hope the code accomplish. To relate this directly to Wasp, it also helps to force you to look at what is happening on the terminal with a more careful eye, as you attempt to follow the events on the screen in order to create that word picture.

But... the first rule is to start out simply. Don't get too fancy, and don't try to do too much with the first draft of a script. I prefer a modular approach to script writing, where one can add features and functions simply by adding a new procedure to a basic script. In this way, a first draft script can be kept quite simple, and then have other routines added to it as they are written and tested.

However, this does not preclude increasing the size of the main (or any) procedure. Once a procedure has been tested and found to work properly, particularly if it is to be called only one time, I will simplify the structure of the script by removing it as a separate procedure. It is then added to the procedure where it was called initially, making it part of that procedure.

Basically, when I begin work on a new script, or on a new procedure to be added to a script, I make sure of two things... 1) that I have lots of paper to rough out in English, flow chart, and code information, and, 2) that I have plenty of disk space available for various versions of the code being tested along with backups of the new code, just in case!

I remember one time when I was away from my computer for the weekend and brought with me a printout of the code I was working on, and a pad of paper to use
when working on the modified code. My daughters complained bitterly that I had to have destroyed three trees writing that relatively small piece of code because I used so much paper writing it. This brings up another routine that I use frequently. Rather than physically testing the code while on line, I often use diagrams, flow charts, and logical analysis to work out on paper, what should happen. This procedure will test the logic of the script before actually compiling it and running the code.

The logic of the script can be supremely important when trying to track down a bug or mistaken action. One of the main benefits of using *English* coding, and flow charting a script is that it tends to improve the logic of that script. Rather than bouncing here and there, from one procedure to another, it becomes easier to write the various procedures and routines in a more logical fashion. This makes the script
easier to follow later, if a problem develops, or you choose to change some section of it. A logical arrangement of sub-procedures and routines within a procedure makes it far easier to locate a section of code should you decide you want (or need) to change.

To that end, I also attempt to use descriptive variable names and procedure/function names, where ever possible. As example, in my PCB Freedom script, the procedure which does the physical dialing of a system and manages the on line functions is called "proc dial_boards". The procedure which it

creates, a dialog box to edit system settings while off line, is called "proc edit_dlgs". The routine to add a new system to the configuration list is called "proc add_item".

I am sure that you can see how this might prove advantageous. On a little script like a basic log on script, this is not of great importance, as the script is fairly short. My GHOST BBS script is currently around 9,500 lines of Wasp code. Locating a particular section within that 9,500 lines can be quite difficult without some *sign posts.* I try to use descriptive variable and procedure names, with a logical connection to what they are for, as my *sign posts*.

I also use the search and replace feature of my editor as frequently as possible. No, I don't use the editor shipped with PCP/Win 2.0... I am a long time Norton DeskTop user, and simply prefer the familiar, Norton's DeskTop Editor. Often times, a variable, or procedure may start life with a particular name, descriptive of it's purpose and place within the logic of the script. However, as the script takes on new features and functions, that descriptive name may no longer be valid, or useful. When this happens, *search and replace* allows for easy change of one name or variable in use into another that is more informative.

The thing to remember is that there are no set rules for methods or procedures to write a good script. What works best for one code maven may be deadly to another. I know my old CompScience Prof. from many years back would probably have a fit about that comment. He constantly emphasized following traditional rules when coding. However, in the real world, I have found that some of those rules don't always work.

That Prof. used to regularly tell us to never use a GOTO label, but to always call a sub-routine instead. Theoretically, this may work, but in the real world, there are many occasions when you do not want the script to return to a given spot, but rather to branch off through a different set of code. GOTO works rather well for this, while calling a sub-routine can be tricky to do the same sort of thing. Since Wasp is neither Fortran nor Cobol, or do I run Procomm on a mainframe, the *rules* he used to drill into us are not necessarily applicable.

There is another thing that I find important when breaking the rules... you just might discover something that helps execute the code! The trick is that, sometimes, what conventional wisdom -*the rules* says can't be done, - just might be possible. But you will never discover these code segments if you follow all of the *rules*. In both FREEDOM and GHOST, there are several bits of code which, when first written, were discarded because examination on paper *proved* that they wouldn't do. But, when all else failed, or the conventional methods grew too cumbersome, I would invariably fall back on the "impossible" code, and generally, found that what appeared impossible on paper, worked quite well when compiled.

ww

Remember that there is a corollary to this "trick. Sometimes what appears, at first test, to work, is really *impossible* and can bite when you least expect it. One pitfall to being a *ground breaker* is that sometimes you find that, instead of breaking new ground, you are over the edge of the cliff with an anvil for a parachute.

Never discount the impossible... fairly early in the beta of FREEDOM, I ran into a problem with some very strange responses being sent by the script. The code was written to delete all characters from the string variable resulting in a null string and nothing being sent. When I discussed it with the folks from Datastorm, I was told that what I claimed to have happening was impossible, and that deleting one by one, the
characters of a string HAD to result in a null string when the last character was deleted.

Further testing, careful observation and notes of what was being sent when nothing should have been sent, showed me that somehow the string, once all characters were deleted, was being assigned a value. That value seemed to be the central six characters from the LAST string variable accessed before the current string variable had it's last character deleted !! Does this make sense?? The solution: When the last character was supposed to be deleted from the string, instead of actually deleting it, I began assigning the system variable $NULLSTR to the string variable. Now, impossible or not, the string variable actually was a null when I wanted and expected it to be one.

In other words, never say never, and always suspect that what is not possible for a script to do, just may be possible. Further, never turn your back on a completed and fully tested script.... they can be mean, vicious and downright despicable!

I suppose that is enough rambling on and lecturing for now... in our next column, we'll go back to dear old George, and help him develop his simple log on script into something more generic, and more useful.

Gregg Hommel is a communications consultant for Delrina. He hosts several of the Procomm conferences and co-hosts the Rime Windows conference. He has been a Procomm beta tester and is the author of the Aspect scripts Freedom and Ghost **.**

ww