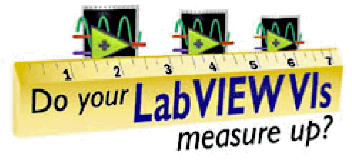
Document Type: Tutorial
NI Supported: Yes
Publish Date: Sep 06, 2006

Rules to Wire By -- Part II

Welcome to Part Two of *Rules to Wire By* -- guidelines for good LabVIEW programming practices. In the last issue of LTR, Part One of this article discussed general programming style and guidelines for creating LabVIEW front panels and user interfaces. Generally everything suggested in that article could be directly applied to any programming environment -- C/C++, Visual Basic, and LabVIEW. However, this article is more LabVIEW focused as it offers guidelines for good LabVIEW programming practices to follow when creating block diagrams. Although I list each suggestion as a rule, don't take this too literally. These items are actually a collection of suggested guidelines, tips, and practices that I recommend using when programming in LabVIEW.



Once you have designed your code and laid out all of your functions, you need to connect them with the LabVIEW wiring tool. How you do this can directly affect the success or failure of your program!

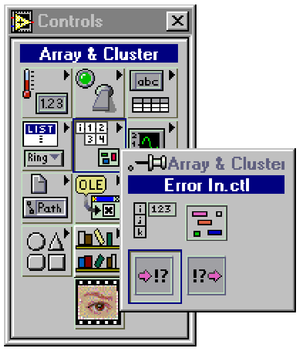
Rule #32 -- Use clean left to right wiring with no hidden wires.
Remember the rule from Part One of this article about consistent connector panes? Here is where they come in handy. If all of your connector panes are constructed and laid out similarly, wiring becomes a snap. The output on the right of one VI directly connects to the input on the right of the next one. If the connector panes have the same number of terminals, no bends in the wire will be needed either.

Never route a wire behind anything! Whenever you do so, you can no longer see all of the connections a wire may or may not have. You also may forget about it if you do not see it. This can be a big problem if you select a couple of items in a loop or case and there is a hidden wire behind it. Even though the wire is not part of the loop you are selecting, LabVIEW selects it. So if you delete or move your code, you will delete or move this hidden wire, with unknown consequences. If you are lucky, you will get a broken arrow -- if not, you will never know it happened (until you try debugging that is).

Rule #33 -- Avoid indiscriminate use of Remove Bad Wires.
Although **Remove Bad Wires** is a handy tool, I almost never use it. Always remember that it will remove any bad wire on the diagram, not just those you are looking at. The other bad wires it deletes may be part of some code you are wiring but have not had a chance to finish yet. Delete them and you are back to square one. More of a problem though, is the wires that get deleted without generating any errors. Think of a shift register that loses the wire to its initial value. You most likely will not get a wiring error, but it is unlikely the VI will work as you expect it to. Think of the hours you might spend finding this one! For those of you with C or Basic programming backgrounds, think of a command in your editor that allows you to delete all syntax errors at once. Sounds sort of dangerous does is not?

Rule #34 -- Use double and triple clicks to look for wiring problems.
Say you are having a wiring problem. What is the best way to resolve it without using things like **Remove Bad Wires**? Try clicking on the wire you are concerned about. Single clicks select a line segment, double clicks select a branch, and triple clicks select the whole wire. A triple click is especially useful for finding all places a wire is connected to, including those you know about and those you do not. One error I have seen that is impossible to correct any other way (short of starting over) is when two input terminals get connected together. Whenever you try to wire the second input, you will get a broken arrow because the terminals have multiple sources. Since the wire is hidden under the icon (see why hidden wires are nasty), you will not see it. Likewise, if you try to delete bad wires, it will not get that little stub between the terminals if one good wire is connected. However, if you triple click, the wiring problem becomes immediately obvious. I can find no better or easier way to make sure I have wired to the correct terminal of a subVI than triple clicking.

Rule #35 -- Use Create Control/Constant as much as possible.
Since the advent of LabVIEW 4.0, it has been possible to create constants on the diagram by simply right clicking and selecting **Create Constant**. Use this as much as you can! Besides its simplicity, it prevents a few nasty bugs from being introduced. For example, suppose the subVI is using an enumeration. If you wire a number into the subVI and later change the enumeration, you will never be the wiser. If instead, you create a constant, not only do you get a nicely documented input, but if the enumeration is ever changed, you will get a broken arrow telling you to check the input. Think of the hours of debugging you can save this way. I can not stress enough keeping your diagrams neat. I find that neat diagrams have fewer bugs, look more professional, are easier to maintain, and were probably written by someone who designed their code before they started writing it. Sloppy diagrams, on the other hand, are usually full of all sorts of nasty unexpected bugs and were written quickly, under duress, or without much thought. I have found that the few extra minutes it takes to make a neat diagram often saves me hours or days in debugging later!
For simple programs, error handling is something we all gloss over in an effort to just get things done. In a larger program, however, it is what separates working and usable applications from buggy and difficult to operate bits of code. All programs will encounter errors -- a well written and professional application will simply deal with them better.



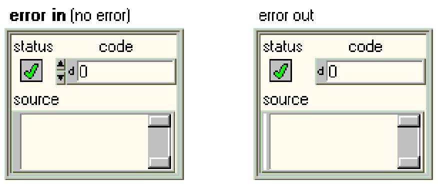


Figure 6: The Array and Cluster Control palette contains built-in Error In and Error Out clusters.

Rule #36 -- Use Error In and Error Out clusters.
Use the Error In and Out Clusters (see ***Figure 6*** above) from LabVIEW's **Array** and **Cluster Control** palette for all VIs that could encounter a problem while executing. This includes all input/output routines involving files, serial ports, GPIB, etc. These error clusters provide an easy and consistent way of reporting error information to a calling VI. They can also be used by the caller to control the flow of execution.

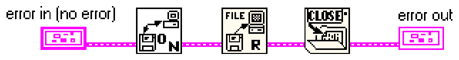


Figure 7: Use error clusters to control the flow of execution.

In the example shown in ***Figure 7***, the Open File must complete before the Read File, and the Close File will wait until everyone is done. Notice how visible everything is -- no sequence structures hiding the data, no intermediate checks for errors to complicate the diagram. Also, if any errors occur prior to this VI, the Error In cluster will keep this VI from executing unnecessarily.

Rule #37 -- Use NaN (Not-a-Number) instead of error clusters where appropriate.
Some VIs, especially math functions, do not need the extra information present in the error clusters. Instead, consider setting the output to NaN when the function fails. There are numerous advantages to doing this. First, any calling VI can use LabVIEW's Not a Number/Path/Refnum to easily see if the function was successful. Second, NaN doesn't appear on graphs. So if you were plotting data and suddenly your data went out of bounds, the graph would show no data for that interval. Although you could use plus or minus infinity for the same purpose, they are not as easy to check for and cause vertical lines to appear on your graphs.

Rule #38 -- Do not show errors too early -- pass them down the line.
When encountering errors, you do not always have to report them right away. Instead, deal with them at the appropriate time. For example, in the file input example above, there is no need to report errors at each point (after opening, reading, and closing). Instead, wait until the end to announce the error. Not only will you spend less time adding error checks, but the operator will not have to deal with as many error messages. Of course, if you have something important to say, go ahead and do so. It may not always be best to wait.

Rule #39 -- Pass errors through unchanged.
If you are creating your own input/output VIs, make sure to pass any incoming errors through without modification. For example, if an error is detected prior to your VI, skip your code and return the error passed to you intact. Do not accidentally overwrite the source or error code, or the error handler may not be able to determine exactly what happened.

Rule #40 -- Think programmatically. Do not always stop or show too many dialogs.
Consider adding a central message display instead of using error dialogs. Not only does this put the errors in one place, but it also prevents the operator from spending all of the time pressing **Ok** or **Cancel**. I like to use a LabVIEW Queue to handle my error messaging. I create the Queue in my main VI and run the **Remove Queue Element** in a separate loop. All my subVIs simply pass their error or status information to the main VI using an Insert **Queue Element**. Not only is this simple, but the use of the queue makes it very efficient as well. One word of warning... To handle errors programmatically, you will need to turn off many of the advisory dialogs that come up as a default in many of the built-in LabVIEW VIs. Not all VIs include this option, so you may have to modify them yourself. (For an example of this, see the advisory dialog input in the help window in ***Figure 8*** below.)

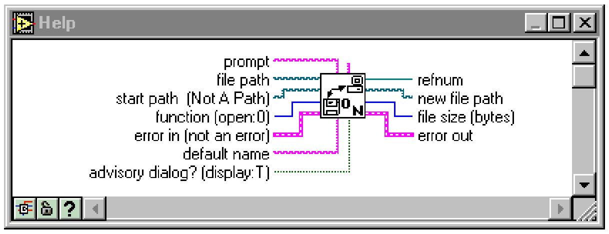


Figure 8: Handling errors in a central message display may require disabling advisory dialogs in built-in VIs.

Rule #41 -- Try to close valid reference numbers, task IDs, etc.
One exception to skipping your code if there is an incoming error is in any **Close VIs**. Regardless of the status of any errors coming in, you should always try to close any handles, reference numbers (refnums), taskIDs, ports, etc. that may have been opened by an earlier VI. This is pretty easy in LabVIEW as there are numerous comparison functions to check to see if the IDs are valid.

Rule #42 -- Register user-defined error codes and use LabVIEW's extra ones.
For many problems, you will find suitable error codes in the error ring on the **Additional Numeric Constants** section of LabVIEW's **Numeric** function palette. But you are by no means limited to just these error codes. LabVIEW provides many more error messages. If you need more than the built-in error codes, you can define your own. For example, I often create a serial driver for a new piece of equipment I am using. There are several handy error codes in the 1200 to 1240 range that I use with the driver. Error code 1240 indicates a timeout, while 1202 indicates that the driver was unable to initialize the instrument. And there are many more -- run a stand-alone copy of the **General Error Handler** to find them. If you still do not find one, create your own using LabVIEW's user-definable codes from 5000-9999. For each code you use, make sure to add an entry for it in the user-defined input arrays of the **General Error Handler**.

Rule #43 -- Use negative error codes for fatal problems and positive for all others.
By LabVIEW convention, if you are unable to complete your task, set your error status bit and make the code negative. Likewise, if the task completed, but generated warnings, make the error code positive. A later VI can look at the code and decide whether or not to continue.

Rule #44 -- Build good error messages.
Use all of the tools at your disposal to generate good, descriptive error messages. The file constant **This VI's path** or **Call Chain** will help you determine exactly where the error occurred. Use **Format Into String** to include not only the current VI's error, but perhaps also the errors of any subVIs or math functions. A good error message will make it easier for the operator to determine what happened. A detailed error message will make technical support easier for you.

Using standard styles and guidelines will result in quicker development and more maintainable, reusable bug-free code. Look for more suggested LabVIEW style and programming guidelines on debugging or more advanced LabVIEW topics in future editions of LTR. In the meantime, try to implement some of these style and programming conventions in your own programs. As an aide, I have included a basic set of VIs that demonstrates many of the techniques mentioned in these two *Rules to Wire By* articles. Look for these example VIs on this issue's LTR resource disk.

As stated in Part One, the intent of this article is to spark discussion in the LabVIEW community, resulting in more LTR articles or letters to the editor on this topic to help promote the establishment and use of LabVIEW standard practices. Remember that the whole point of presenting these suggested guidelines is to get you to think about how you program. You may disagree with some of my suggestions, or these suggested guidelines may not be practical for your programming environment. The actual style or standard guideline is not so important as is the fact that you are actually using a standard. Choose what works best for you or your programming team and stick with it.

About LTR
LabVIEW Technical Resource(TM), LTR, is the leading independent source of LabVIEW-specific information. Each LTR issue presents powerful tips and techniques and includes a resource CD packed with VIs, utilities, source code, and documentation.
About Stress Engineering Services
Stress Engineering, a National Instruments Select Integrator, develops custom applications using many of National Instruments products, including LabVIEW.

Related Links:
Stress Engineering Services