Rules to Wire By -- Part I

Publish Date: Dec 10, 2014

Table of Contents

1. Overview
2. How do you define "good" LabVIEW style?
3. Sources of Information
4. General Guidelines
5. Naming Conventions
6. Front Panels
7. Regular VI or User Interface?
8. User Interfaces
9. Regular VIs
10. Icons and Connector
11. Fonts and Colors
12. Summary
13. About the Companies

1. Overview

As with any programming language, especially one that is used in as many different countries and on as many different computing platforms as LabVIEW, there's no substitute for good programming style. Style helps programmers write better code that is easier to use, maintain, and review. Even with all of our differences (programming experience, language, color preferences, etc.), a good style guide offers standards that promote reuse and sharing of code and programming concepts, ease of documentation and ease of code maintenance.

2. How do you define "good" LabVIEW style?

Now of course things are not really that simple. Style can be very subjective. As a matter of fact, in reviewing the references to good LabVIEW programming practices that are available, I found a few items that I disagree with. It is not that I do not believe in them -- we all want to make nice VIs with detailed documentation and full-color icons. It is just that many of us simply do not have the time, in practicality, to completely follow the guidelines. Part One of this two-part article suggests LabVIEW style and programming guidelines for the front panel, user interfaces, icons and connector panes. Part Two, printed in the next issue of LTR, will cover programming guidelines for the block diagram and more LabVIEW advanced techniques. The suggestions I present in this article are a combination of the following:

- Desired styles recommended by National Instruments and the LabVIEW documentation,
- Consensus in the LabVIEW community at large, and
- My experiences in nine years of LabVIEW programming.

For example, much of the information about developing user interfaces comes from books on the subject or feedback I have received from various operators. This article also adopts and adapts several ideas from material provided to me by Gregg Fowler. Special thanks to Gregg for his ideas and base material.

Style can also differ greatly from country to country, company to company, or work group to work group -- and there is absolutely nothing wrong with that! What is most important is that you and your coworkers select a standard LabVIEW coding style that suits you and stick with it.

3. Sources of Information

This article is by no means complete. There are many references to good LabVIEW programming practice available. Several years ago, Meg Kay and Gary Johnson co-wrote a short article called the LabVIEW Style Guide. This article offered guidelines to follow when developing LabVIEW programs. By using these techniques, you were more likely to create more maintainable and robust LabVIEW code. However, many of those guidelines were written for LabVIEW 3.x, whereas we now have LabVIEW 5.1. With features such as Active X, language translation tools, documentation tools, etc., we desperately need a style guide update.

Meg and Gary's original style guide was the foundation for much of what you will now find in the LabVIEW Professional Developer's Kit.

There is another older but still very applicable document that we have included on this LTR issue's resource disk. The Windows 95 User Interface Style Guide was written by Chris Roth around the time Windows 95 began shipping. It detailed how to update your user interfaces to include features like recessed controls, tabbed dialogs, etc. It also included a very useful LabVIEW library containing many new controls and several example VIs.

Some other available LabVIEW programming references are National Instruments Application Notes and Technical Notes. For example, Technical Note Number 111 is actually a list of standards for instrument drivers, but the overall intent can be applied to other VIs.

Also, consider more general publications. I highly recommend a book written by Steve McConnell of Microsoft called Code Complete. It covers a wide range of programming topics like requirements, design, debugging, code reviews, and documentation. This book is geared towards C programmers, so the LabVIEW programmer will need to do a little picking and choosing of the topics and decide what to apply. There are also books available from Microsoft and Apple covering the user interface guidelines for their operating systems. Similar ones should be available for other platforms.

As a final point, do not underestimate your own feelings on style. If something "feels" right or wrong to you, then it probably is.

Related Links:

- LabVIEW Professional Development System for Windows
- LabVIEW Professional Development System for Mac
- LabVIEW Professional Development System for Sun Solaris 2
- LabVIEW Professional Development System for HP-UX
- Code Complete

4. General Guidelines

So, let's start with the basics. You can use these suggestions on any project, large or small, simple or advanced. Although Part One of this article is intended to focus on guidelines for LabVIEW front panels and user interfaces, these first set of rules are general programming philosophies or guidelines that apply to all aspects of LabVIEW, including your block diagram code.

Rule #1 -- Design first, code second.

I know you that have heard this tip numerous times before, but that's because it is such a good one. Sit down with a few sheets of paper and write what your users require the program to do. Also, draw out what you want your user interfaces or panels to look like. **Most importantly, don't even think about LabVIEW during this step!** In essence, you are creating a Software Requirements Document. The important thing is to know what you ultimately want to achieve.

At this point you may want to mock up the screens and run them by other people in your group before moving on. You might want to go directly to prototyping interface panels using LabVIEW controls and indicators. One of LabVIEW's strengths is in its use as a quick prototyping tool. There's nothing saying that the front panels have to work or have complete diagrams at this stage. Prototyping the front panels can help get the kinks worked out of the user interface up front and save you some coding and debugging down the road.

Next start thinking about data structures, program flow, events, states and state diagrams. This is how you expect the code to work together (i.e. if this happens, then do this next, but if this happens, then do this instead). Try to think of your application in terms of the dataflow concepts that LabVIEW is based upon. Several good articles available in past issues of the LabVIEW Technical Resource demon-strate examples of LabVIEW architectures for event-driven applications. Try looking at:

- Architecture: The Big Picture of a Graphical Program by Jeff Parker in the Winter 1995 issue (LTR Volume 3, Number 1)
- State Your Case! by Lynda Gruggett in the same issue.
- Interactive Architectures Revisited by Gregg Fowler in Spring 1996 issue (LTR Volume 4, Number 2)

Finally, keep in mind the time scale and budget for the project. If you are doing something quick and dirty, it's acceptable to streamline this process to start coding sooner. If you are building software that will be maintained over the long term (or has any chance for that,) do more design up front. Thinking up front pays off down the road when you are enhancing for version 2, 3, and on.

At this point, we are almost ready to start writing code. Part Two of this article will cover the LabVIEW software development process in more detail with tips on designing for modularity and creating clean APIs, or calling interfaces, between VIs.

Rule # 2 -- Try code snippets before using them.

I like to make sure my code will work before writing it. To do this, I take a small section of that code and try it out in a test VI. That way, I make sure I have working code before it gets buried where it is hard to find and debug later on. For example, I have been programming LabVIEW for years, but I still can not remember whether to use **Threshold 1D Array** or **Interpolate 1D Array**. So, I code a quick test VI to find the correct one before including that code in my block diagram. Following this rule will save you debugging trouble down the road.

Rule #3 -- Plan for reuse

Of course, now that you have written this bit of working code, don't just throw it away -- save it! You might need it again on this or the next project. These also make great examples for new coworkers or anyone else who may need to take over your code in the future. I used to never save test VIs and found myself spending a lot of time recreating them. Considering the cheap cost of

megabytes and gigabytes nowadays, there is just no reason not to save. Therefore, save test VIs where appropriate. Make VIs that can stand alone and be reused elsewhere. You should keep notes on what is clever and useful (such as particular control and indicator combinations to achieve special effects).

Rule #4 -- Keep it clean

No matter what else you do, try to keep your code neat and clean. Not only does this make your code easier to follow for the next person (be warned -- may be you), but neat code is usually good code. In the act of keeping things clean, you will find that you are less likely to make mistakes or forget something. I don't know how may times I have noticed wiring problems while cleaning up my diagrams. I will go into more block diagram details in Part Two of this article in the next LTR issue.

Rule #5 -- Comment, comment, comment

As LabVIEW programmers, we are all very busy (trust me, I know). But there is NO substitute for good comments in all of the code that you write. Comment all tricky code, tough algorithms, custom utilities, and so on. As mentioned in Code Complete, "Good comments don't repeat the code or explain it. They clarify its intent. Comments should explain, at a higher level of abstraction than the code, what you are trying to do." Remember that the person who looks at this code six months from now may be you! Anything added now will make that time go so much smoother.

What I recommend is that you write a brief (usually no more than one or two sentences) description of each VI in its VI Info screen. Then add descriptions for any unusual controls or indicators. If you document controls and indicators in their description boxes, the user can see those in the floating help window as he moves the mouse over the various controls. It's great if you can document them all, but in today's busy environment, there may not be time for that. If you are disciplined enough to document your controls, indicators, and VIs, then you have the added benefit that programming documentation can be automatically created using LabVIEW's Documentation Tools.

On the diagram, label all code that isn't obvious. At a minimum, try to label all structures (while loops, for loops, cases, and sequences). That way the main portions of the code are always documented. A very handy feature in LabVIEW is string and enumerated cases. These handy structures are pretty much self-documenting. In the left side of the example in Figure 1, you have to add extra comments to describe each case. With the string case shown on the right and good string names, that task takes care of itself.

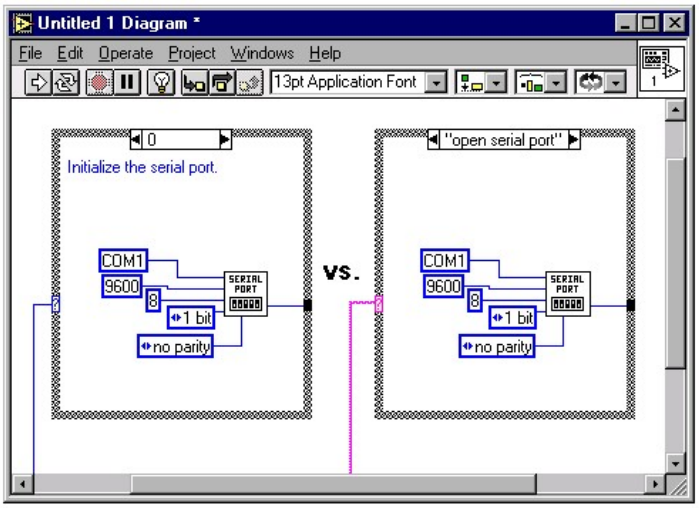

Figure 1: Use strings or enumerated types to create descriptive names for case structure frames.

Also include any references that you use. For example, say you have a piece of code that converts a dewpoint to its corresponding partial pressures of water value. If you obtained that formula from some special source, then reference that source as a free standing text block on the diagram as shown in Figure 2.

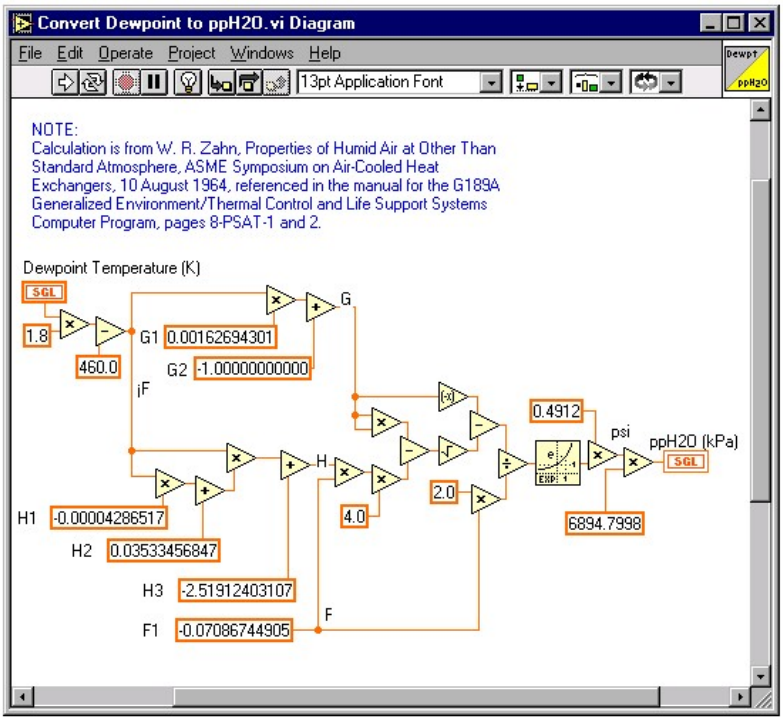

Figure 2: Use free standing text boxes to document references such as formulas or algorithms.

5. Naming Conventions

Naming conventions really fall into the general practice category, but they are so important that I decided to emphasize them in their own section. Of course, the most important naming convention of all is:

Rule #6 -- Use descriptive names.

Do not use names like **VI #1.vi** or Untitled.vi when something like **Save Data** to **Spreadsheet File.vi** or **Convert Voltages to Engineering Units.vi** will do the better job. Try to include action words like **Open, Close, Save, Calculate, Update,** etc. for subVIs that do something. Use words like **Display, Panel, View, Screen,** etc. for user interfaces.

I like to prefix my VI names with a category or VI function designator. This is useful if you ever transport your VIs using LabVIEW libraries (**.LLB's**). It is easy to get a hundred or more VIs into a library, but sorting those hundred out at the destination is often tough. By prefixing those VIs with words like **Display, File, DAQ,** etc., it is easy to group and move them. I also use this to separate my reusable VIs from those that are project specific. VIs I might use again will start with **File** or **Utility** while the project VIs will usually start with the project name or client. This makes my VI management easier because I can tell at a glance what a VI's type or function is.

Also, don't forget to include the "**.vi**" at the end of the file names. Many operating systems like Windows and Unix use these extensions to determine what files belong to what application. Of course this brings us to the next rule...

Rule #7 -- Watch for platform naming problems.

If you know your VIs will eventually end up on a Macintosh, try to keep them 31 characters or shorter in length. Also try to avoid characters like "\:/?*<>#" that some file systems use for special purposes. Even though you may not expect to be changing platforms, someone else may find your VI useful and need to use it in their system.

6. Front Panels

Now let us move from generic style guidelines to ones specifically for LabVIEW front panels and user interfaces.

7. Regular VI or User Interface?

Before you do anything with a front panel, decide whether it will be a regular VI or displayed to the operator as a user interface panel or dialog. Each will need to be treated differently.

Rule #8 -- Use standard LabVIEW controls for regular VIs.

Use dialog controls where appropriate for user interface VIs. For simple VIs that only a programmer will see, use all of the LabVIEW controls available to you. Keep your screens neat, but don't worry too much about cosmetics. However, if you're making a user interface panel, use the dialog controls where appropriate, including the controls within the attached Win95 library. If you are a Macintosh or other platform user, try to make your user interfaces look appropriate for the platform.

Rule #9 -- Use small letters for controls on regular VIs.

Capitalize the first letters of user interface controls. Again, use the capitalization and font guidelines that match the platform you are using. I often pop over to a few other applications on my computer to see how they set up dialogs and user interfaces. I also try to use button labels similar to the ones used by my operating system. And don't forget localizing the look. For example, English

versions of Windows capitalize the first letter of button labels while Spanish versions do not. Always try to be consistent with the user interface guidelines for your platform and language. If applicable to your application, make use of the built-in Localization features in LabVIEW.

8. User Interfaces

Rule #10 -- Group controls logically.

If you have controls on a panel that are related to one another, group them by putting them in a cluster control or wrapping a decoration around them. If the data are used together programmatically, I use a cluster. If they are not, I use one of the decorations that matches the operating system, platform, or country I am writing code for. See David Moschella's LTR article titled **A GUI Machine** (LTR Vol. 6, No. 1) for more examples.

Rule #11 -- Watch your screen sizes.

Make sure that your panel can be displayed on the system you intend to use it on. If you are not sure then you may want to use 640x480 or 800x600 as default. You can also use LabVIEW 5.1's screen sizing features. Likewise, ...

Rule #12 -- Do not space controls too closely (especially when using touchscreens).

To make things easier on the eyes and fingers, try to leave some gray or white space in between the various controls on your front panels. If you are using a touchscreen, figure that the smallest anything can be is about 1/2" square. If things are any smaller than this or any closer together, you will have a hard time clicking on them without hitting adjacent objects. Also, ...

Rule #13 -- Don't place consecutive dialog buttons on top of each other.

Say you have a dialog that asks the user if they want to rename or replace a file. Suppose they choose **Replace** and another dialog appears that asks them if they are sure they want to overwrite the file. If you place the **Replace** button in the same general location as the **Cancel** or **OK** buttons on the next dialog, it will be very easy to double click or hit a button unintentionally (trust me, I've done it). Instead, move the next dialog window or buttons so that the operator can not accidentally select overlaid buttons.

Rule #14 -- Use Key Focus or key navigation for the default choice.

If you want the safe or default function to be executed if the operator presses return or escape, use the **Key Focus** attribute or **Key Navigation** setting for that control. This is especially important for dangerous operations like overwriting files, where you do not want the operator selecting something unless they are sure that is what they intended to do. You can make this easier by

Rule #15 -- Use specific words for button names.

Instead of **OK** and **Cancel**, use words like **Save, Replace, Quit, Start, Stop,** etc. for button labels. This makes the program much easier for the operator to use. And, to make things even easier on the operator (although harder for the programmer),

Rule #16 -- Always include a Cancel or Back option.

If the user feels he can press anything on the screen without causing any problems, he will be much more comfortable using the program. Users will also tend to explore and "self-train" themselves when they know they have a safe way to back out. Remember Undo?

9. Regular VIs

Rules #17--19 refer to regular VIs whose front panels are not displayed to the operator as a user interface or dialog.

Rule #17 -- For Booleans, the name should give an indication of the meaning of the true state.

Make it obvious what a Boolean switch does when true. For example, **Reset, Initialize,** and **Cancel** indicate that they will do just that if TRUE. Try to avoid Boolean names like **Don't display dialog** or **No Replace**. Use names such as **Display Dialog** or **Allow Replace**, and switch the subVI's internal logic instead. I would also recommend naming Booleans using **is** or **has** in front and **?** at end where this makes sense. For instance, **Is Scanning?** rather than **Scanning** or **On**.

Rule #18 -- Lay out controls and indicators as they are in the connector pane.

Put all of your inputs on the left and outputs on the right, just as in the connector pane. Put your taskIDs and reference numbers (refnums) in the upper left and right, and error clusters in the lower left and right. Note that this rule only applies to non-interface VIs. Do not worry about this rule for user interface VIs since a logical, intuitive, and efficient layout is more important.

Rule #19 -- Add default values in parenthesis where applicable.

Anytime a subVI control can use a default value, include that value in parenthesis. Not only will that make the default value obvious, but anyone using the calling VI will see the default listed in the help window for that VI. For example, if you have a Reset input, you might to label it as **Reset (F)** indicating that Reset will not be chosen unless the caller wires in a True. If the default is not obvious, use a more descriptive label like **open mode (0:read/write).** If units are important, you will want to include those too -- like timeout **(500 ms)**. Again you are trying to make the programmer's and operator's job a little simpler.

10. Icons and Connector

Panes Moving up the front panel a bit, we come to the icon and connector pane. Here is one area you can definitely make much easier to use.

Rule #20 -- Use the same connector pane, even if it means unused terminals.

Select a terminal layout and stick with it. For example, I use the 12 terminal connector pane in almost all of my VIs. By using the same connector, it becomes very easy to wire icons together neatly. I can also add or delete inputs and outputs without having to re-link to the subVI in all of the calling VIs. If I also use the same connector layout each time, I will know at a glance what a

terminal probably does without even having to look at it. Although you can pick connectors with more than 12 terminals, try not to do it. If you need more than that, consider grouping controls together into clusters instead. You can also make the subVI into several subVIs that each need a little less data. Another connector pane suggestion is to follow the NI style where appropriate for flow-through parameters. For example, refnums and taskIDs flow through along the top connectors and error clusters flow through along the bottom.

Rule #21 -- Use required, recommended, and optional settings for terminals.

When I have a VI that absolutely must have an input wired, I use the "This Connection is Required" option on the connector pane. That way anyone (including me) who tries to use the VI without wiring the required input first will see a nice big broken arrow. I don't know how many hours of debugging I have saved with this seemingly simple trick. Likewise, if there are parameters that I normally do not want changed, I use the "This Connection is Optional" setting. That way the terminal does not even show up in the standard help window, meaning the user probably will have to know what they are doing before using it.

Rule #22 -- Use the "Small Fonts" font for icon labels.

There is a very handy font for creating icon text with LabVIEW. Double-click on the text tool (the giant A) in the icon editor and select Small Fonts from the font list. If you also choose plain style and size 10, you'll be able to label icons in no time. Even if you do nothing else...

Rule #23 -- At a minimum, create a text icon.

We all want to have nice graphical icons. However time restraints often prevent us from doing so. There are times when it's appropriate to take the time to make intuitive pictures, such as when creating icons for instrument drivers designed to National Instruments specifications, but it's generally not practical in most contract situations. In the case where programming time (or budget) is limited, at least put something like "save data" or "convert image" in as text on the icon. A block diagram covered with subVIs that all look the same is hard to review or debug. If you have the time, by all means, create an icon for the VI.

Rule #24 -- Always include a black and white copy of the icon for printing.

Although most of us have nice color monitors, do not forget to create a black and white copy of the icon for printing. LabVIEW often tries to create its own black and white version of the icon, but it is just not the same.

11. Fonts and Colors

Rule #25 -- Use LabVIEW's default fonts and colors.

In creating all of your panels and diagrams, try to use LabVIEW's built in fonts as much as possible. These include the application, dialog, and system fonts. LabVIEW maps these to comparable font families on different platforms, making cross platform VI creation a little smoother. If you do use other fonts, remember that they might not always be present on another computer. If they aren't, LabVIEW will substitute the closest match, usually producing fair, but not perfect results.

The same thing can be said for colors. LabVIEW will map its basic colors pretty well regardless of the hardware or platform capabilities. Choose any colors outside of these and you may have some portability concerns. Of course you also have to remember that color preferences are highly user dependent. To avoid problems, I usually create nice boring gray panels or give my operators control of their colors through a setup screen. It is easy to store color preferences for each operator in a text or configuration file. Then when you run your program, load those preferences and use attribute nodes to color the screen and controls. It may sound like a bit of work, but you only have to do it once. And it sure beats revising interfaces time and time again.

12. Summary

Look for Part Two of this article in the next LTR issue for more suggested LabVIEW style and programming guidelines. Part Two will cover guidelines for the block diagram and more advanced techniques. You are likely to disagree with some of my suggestions, or these suggested guidelines may not be practical for your programming environment. Hopefully, this will spark discussion in the LTR community, and more LTR articles or letters to the editor on this topic to help promote the use of LabVIEW standard practices. Remember that the whole point of a style guide is to get you to think about how you program. The actual style standard is not so important as is the fact that you are actually using a standard. Choose what works best for you or your group and stick with it. Using standard styles and guidelines will result in quicker development, better code, less debugging, and happier operators.

13. About the Companies

About LTR

LabVIEW Technical Resource(TM), LTR, was the leading independent source of LabVIEW-specific information. Each LTR issue presented powerful tips and techniques and includes a resource CD packed with VIs, utilities, source code, and documentation. **LabVIEW Technical Resource** is no longer active.

About Stress Engineering Services

Stress Engineering, a National Instruments Select Integrator, develops custom applications using many of National Instruments products, including LabVIEW.
Stress Engineering Services

Related Links: