Top 5 LabVIEW Rookie Mistakes

Publish Date: Nov 03, 2014

Table of Contents

Although NI LabVIEW software has long helped engineers and scientists to quickly develop functional measurement and control applications, not all new users follow LabVIEW programming best practices. LabVIEW graphical programming is relatively unique in that a lack of adherence to coding best practices is quickly evident with a glance at a user's application. Some users make these mistakes because they truly don't understand the rules behind the flow of data on a LabVIEW diagram, while others are just unaware of features designed to improve the quality of a LabVIEW program.

This article reviews the most common programming mistakes made by inexperienced LabVIEW programmers, along with recommendations on the correct LabVIEW programming approach to use instead.

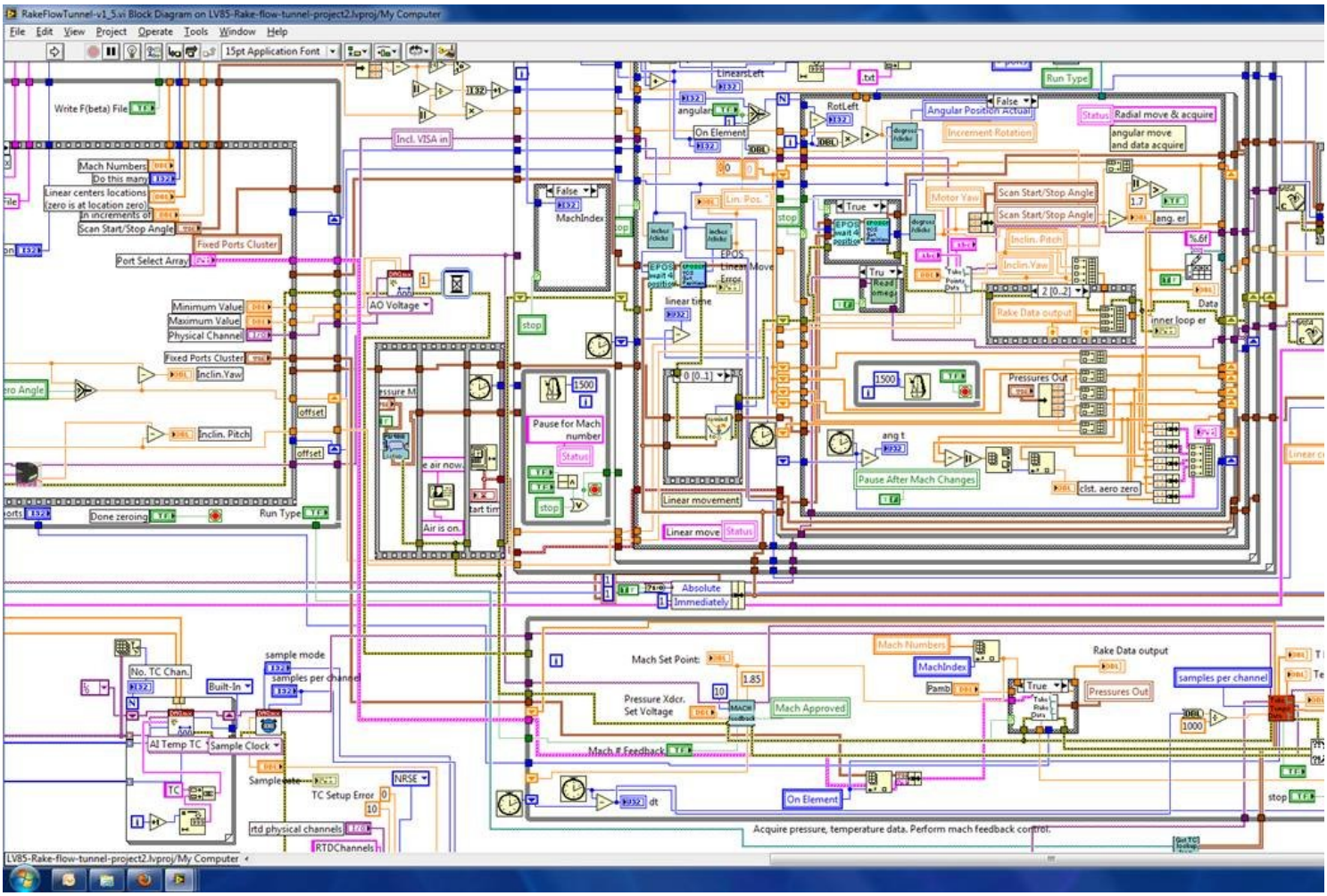

Figure 1. Code full of LabVIEW "rookie" mistakes

1. Overusing Flat Sequence Structures

Many new LabVIEW programmers do not fully understand the concepts behind "dataflow" execution, which is fundamental to LabVIEW programming. One indication of this is that users often overuse the flat sequence structure on their block diagrams. Users often rely on flat sequence structures to force the serial execution of code on the block diagram, instead of using the flow of data with wires between nodes.

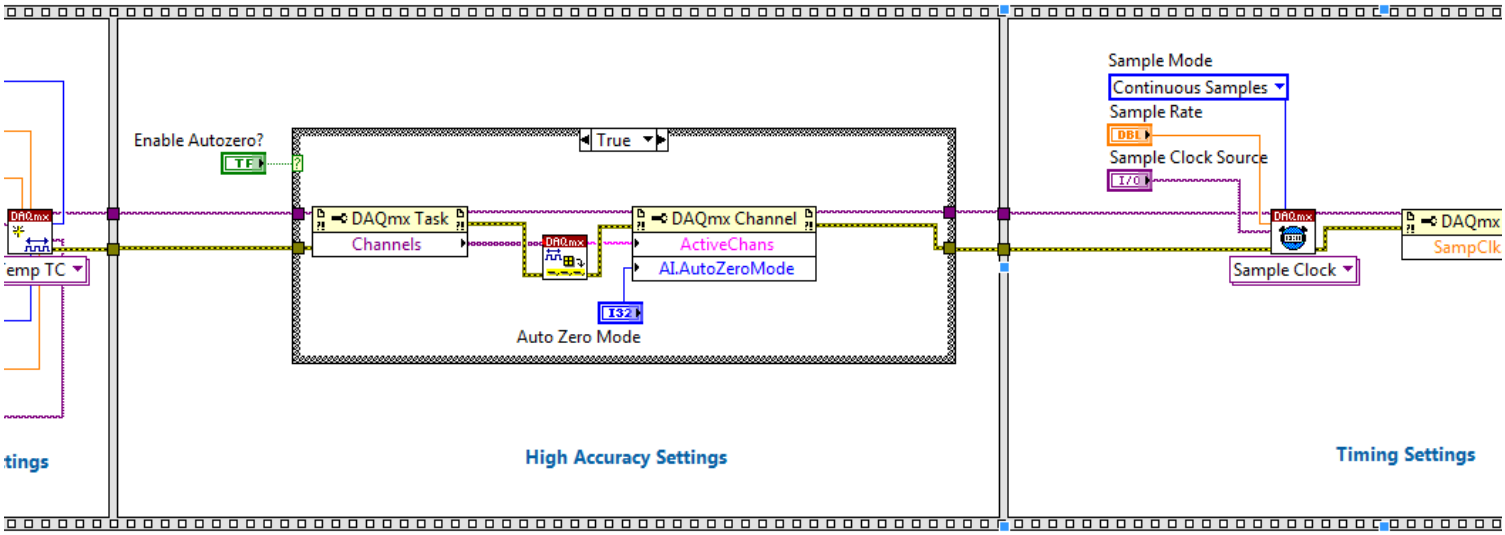

Figure 2. Users often rely on flat sequence structures rather than developing a full understanding of dataflow programming concepts.

Dataflow programming means that a node on the block diagram (subVI, primitive, structure, and so on) won't execute until all the needed data inputs are present. This benefits LabVIEW programmers because independent processes are natively set up to run in parallel, while imperative languages require extra setup for parallel execution. As computers continue to add more CPUs, LabVIEW automatically offloads parallel processes and gains code performance without any extra coding by its users. Forcing execution on the block diagram by overusing flat sequence structures can constrict parallelization and take away this benefit. Limiting unnecessary structures on the block diagram helps with overall readability and keeps diagrams smaller as well.

Error wires are a good way to force data flow on the block diagram, rather than relying on flat sequence structures, and they also benefit an error-handling strategy.

When Should I Use a Flat Sequence Structure?

Forcing execution with a flat sequence structure is useful for benchmarking code performance. By using a flat sequence structure with a tick count inside its frame, you can determine the time it took to execute code in between two tick counts. This cannot be achieved through normal dataflow execution.

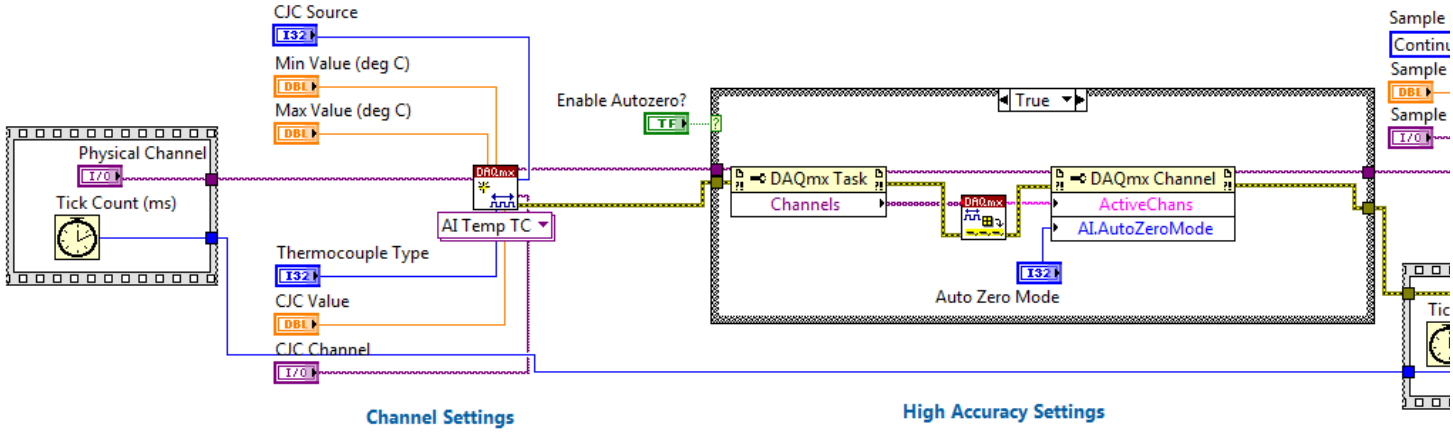

Figure 3. Using flat sequence structures and tick count VIs is useful for benchmarking code.

For more information on dataflow programming, access the self-paced online training (ni.com/self-paced-training) for LabVIEW Core 1 on data flow. Self-paced online training is free with every LabVIEW purchase or for users currently on the Standard Service Program (ni.com/ssp).

Read more about LabVIEW Rookie Mistake No. 1.

2. Misusing Local Variables

Another common mistake in LabVIEW programs is an overuse of local variables. Local variables are a piece of shared memory used to pass data between different sections of a computer program. Commonly used in text-based languages, variables can be very empowering, but can also lead to problems when a race condition is encountered.

Unlike other programming languages where passing data through variables is essential, LabVIEW provides the dataflow method of moving data from one part of a program to another. The parallelism inherent to LabVIEW makes overusing variables problematic because shared memory is often accessed by different code locations at the same time. If this happens, one read/write operation wins the “race” and the other loses. The losing data operation is forgotten, so overusing variables in LabVIEW can ultimately lead to lost data.

You can safely pass data from one part of a LabVIEW program to another using a variety of methods, including wires, queues, events, notifiers, functional global variables, and more. These mechanisms are each designed for a specific use case, but all have the advantage of eliminating race conditions.

For more information on proper techniques for moving data within a LabVIEW program, access the self-paced online training (ni.com/self-paced-training) for LabVIEW Core 1 on Local Variables and LabVIEW Core 2 on Notifiers, Queues, and Events.

Read more about LabVIEW Rookie Mistake No. 2.

3. Ignoring Code Modularity

Often, new LabVIEW users create “fire and forget” applications to accomplish simple tasks and don’t think about how they can use that code in the future. As users start to program more often, they can find themselves rewriting the same piece of code over and over again. You can save yourself a lot of development time by creating modular subVIs out of portions of your code that can be reused within other applications.

If you know that a particular part of code will be reused again within the same application, or have a hunch that it might be reused in a future application, then take the extra time to turn it into a subVI. To prepare a portion of code to be a subVI, the main things that you need to do are add documentation, use the connector pane, and disable some VI properties. One of the easiest ways to start creating a subVI is by highlighting a portion of code on a block diagram and then selecting Edit>>Create SubVI from the menu bar. This scripts the process of putting that portion of code into a separate VI and using the connector pane. You will still need to edit the icon to something that makes sense, add documentation to the block diagram and VI Properties, and turn off some VI settings, but Edit>>Create SubVI is a great starting point to code modularity.

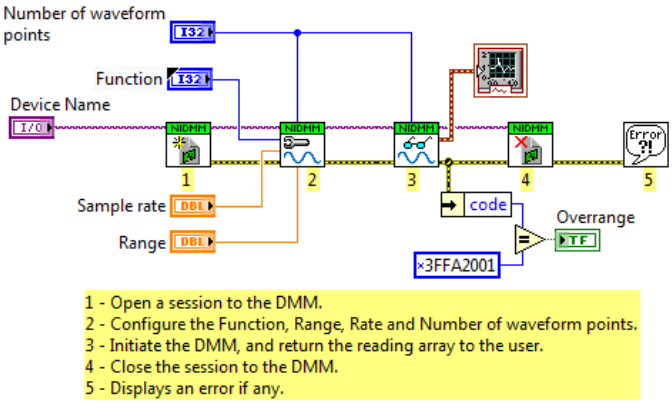

Figure 4. You can save yourself a lot of development time by instituting proper LabVIEW code modularity practices.

The one VI setting that you should make sure to turn off when preparing code for reuse is Allow Debugging. This option is found within the VI Properties (File>>VI Properties) under the Execution category. After your code is completely functional and no longer needs debugging capabilities, like Highlight Execution, turn off Allow Debugging in the execution settings and run your VI again. This is beneficial because more optimizations take place during the compilation process so your application may run faster and the VI will have a marginally smaller physical size on disk because the extra code that enables debugging is taken out.

For more information on code modularity, access the self-paced online training (ni.com/self-paced-training) for LabVIEW Core 1 on Understanding Modularity.

Read more about LabVIEW Rookie Mistake No. 3.

4. Creating Massive Block Diagrams

Many new LabVIEW users have block diagrams that can become huge. Some applications are complex and you can’t avoid having a large diagram, but it also can indicate lack of a programming architecture. Without an underlying architecture, it’s difficult to maintain a program over time and it can be costly to add new functionality later. Just like constructing a proper frame builds a structurally sound house, a good programming architecture provides a safe and secure framework to build your application from.

Software architectures are common frameworks that almost all programmers find useful. Many of the architectures within LabVIEW, such as producer/consumer and state machines, are similar to those found in other programming languages.

Understanding LabVIEW architectures reduces development time and improves application scalability. LabVIEW 2012 made understanding architectures even easier by including templates and sample projects in the release. Templates explain different architectures and when they should be used. Sample projects are larger applications built on top of the templates and demonstrate how templates can be used to meet real-world challenges. You can plug hardware into a sample project and use it as a turnkey application if needed, but it's still open and well-documented so you can customize it for your specific application.

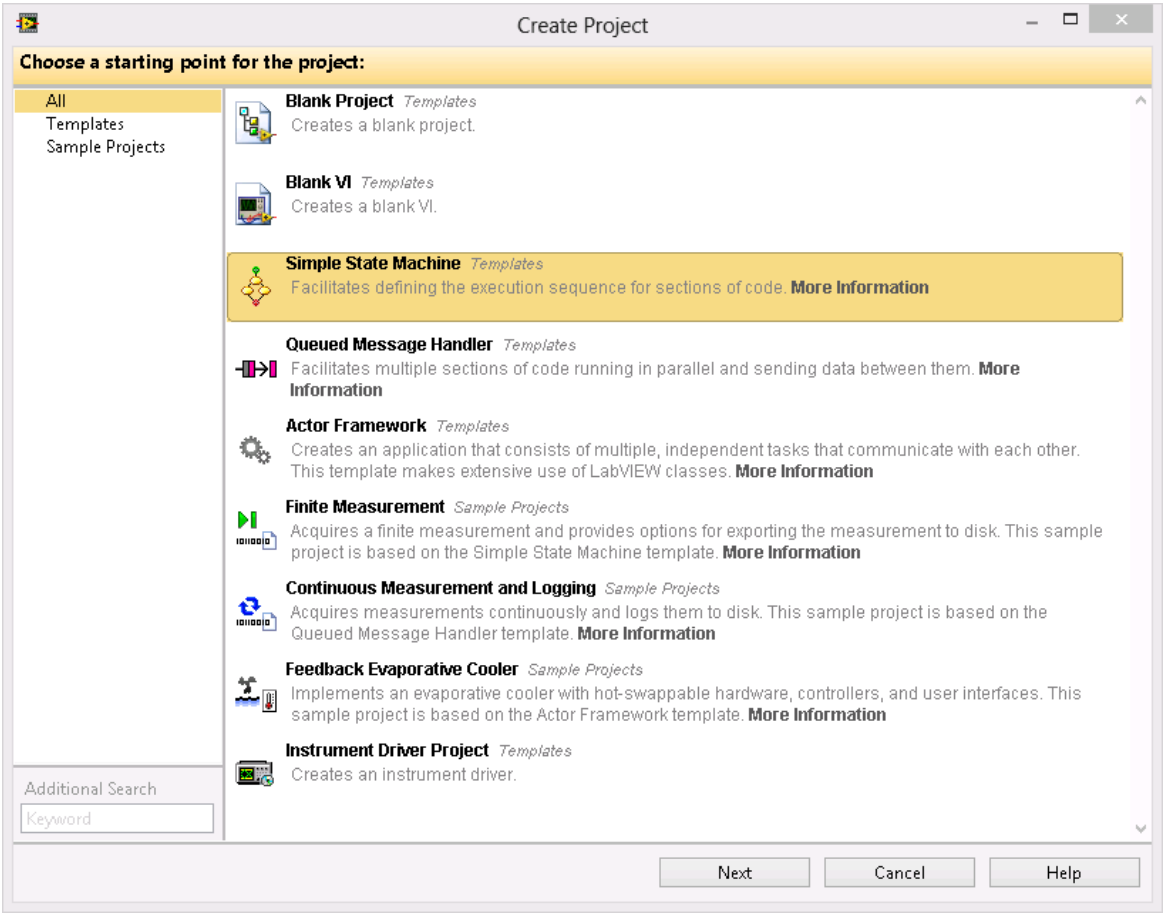

Figure 5. Templates and sample projects available in LabVIEW 2012 make understanding software architectures easier than ever.

For more information on LabVIEW architectures, access the self-paced online training (ni.com/self-paced-training) for LabVIEW Core 2 on Design Patterns.

Read more about LabVIEW Rookie Mistake No. 4.

5. Disregarding the Need for Documentation

Trying to discern what a program written by someone else does is helped greatly by good code documentation. Unfortunately, documentation is normally left until the end of the development cycle after the functionality is complete. This leaves little time to document code properly. Instead, time should be carved out during development to start the documentation process. Documentation greatly benefits the person who wrote the code when they revisit it later and can't remember what mindset they were in when choosing certain code aspects. Coffee and late night programming, a common trait in programmers, can lead, jokingly, to temporary memory loss. Documentation can help programmers get those memories back.

In general, the graphical nature of LabVIEW makes reading code easier than text-based programs, but good documentation can reduce the time needed to "decode" a program even more. The easiest way to add documentation to your block diagram is by using free labels. You can add these by double-left clicking in an empty space on the block diagram and entering some text. Then use an arrow decoration to point to the specific code referenced by the free label. Add pictures by copying them onto the clipboard and pasting them onto the block diagram. Images of physical systems or math formulas can help users formulate code context within the block diagram.

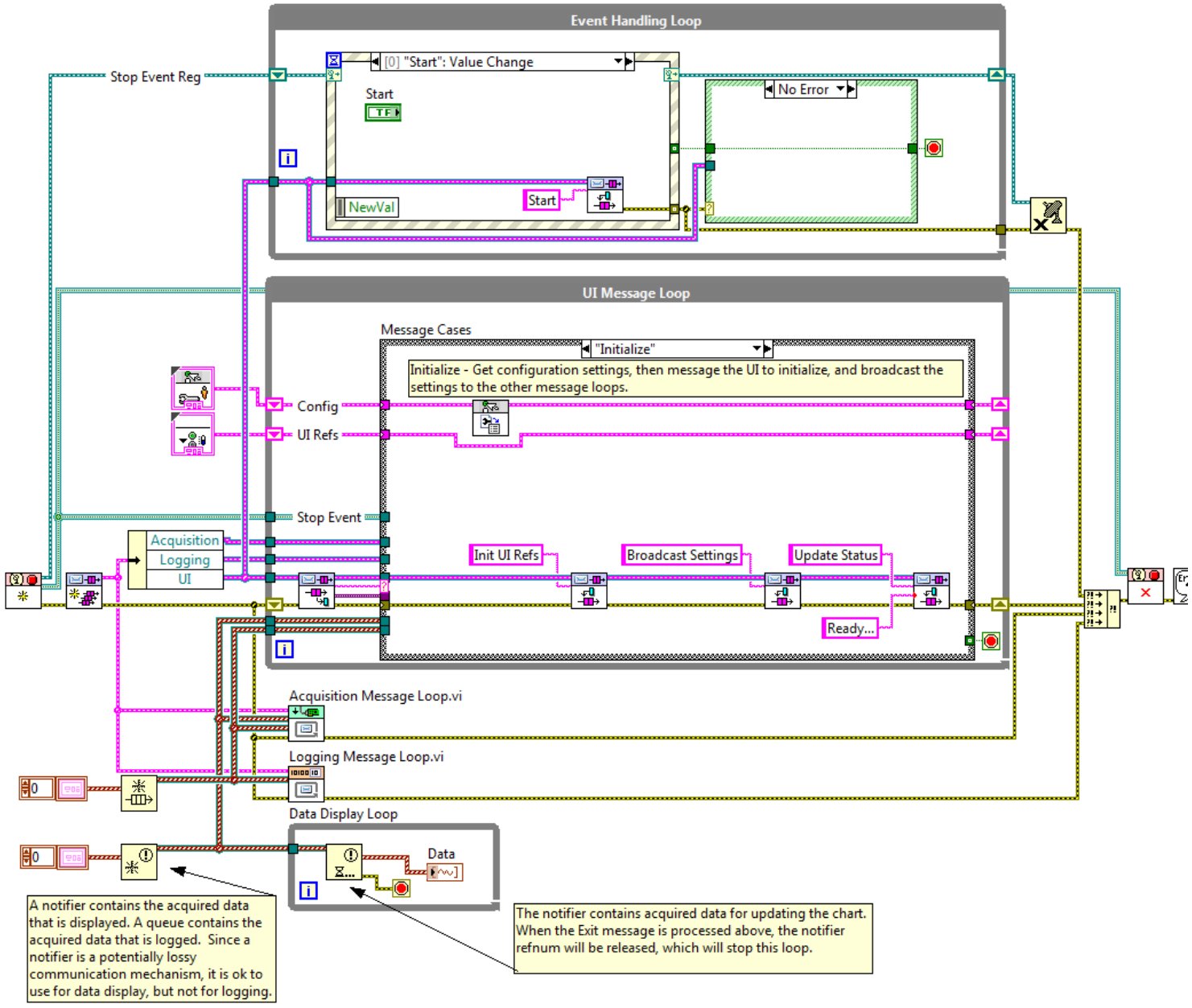

Figure 6. Properly architected and documented code helps others using your code, but also helps you better understand your own code.

Documenting code is not just for reuse libraries, it should be done for every program. Humans learn a lot more about a subject when forced to teach it to others. Writing documentation essentially forces this teaching process and can help programmers understand their own code better.

For more information on code documentation, access the self-paced online training for LabVIEW Core 1 on Documenting Code.

Read more about LabVIEW Rookie Mistake No. 5.

LabVIEW was built to make engineers and scientists more successful at tackling the world's tough challenges. The benefit of having a large programming community of engineers and scientists is that they like to share their knowledge with others. If you have your own LabVIEW rookie mistake that you would like to share, please add your voice by visiting bit.ly/lvrookiemistakes.

For a more in-depth look at each of these LabVIEW rookie mistakes in future articles, stay tuned to NI News.

Customer Reviews

9 Reviews | Submit your review