



ni.com

Five Techniques for Better LabVIEW Code

Peter Blume

President/Chief Engineer, Bloomy Controls

ni.com



This presentation provides five techniques to improve your LabVIEW programs.

Overview

- Introduction
- Write a functional specification
- Use proper data flow
- Implement a state machine architecture
- Use proper error handling
- Document your source code
- Conclusion

ni.com



We will focus on LabVIEW programming and documentation techniques. The specific topics include specifications, data flow, state machines, error handling, and documentation.

Introduction

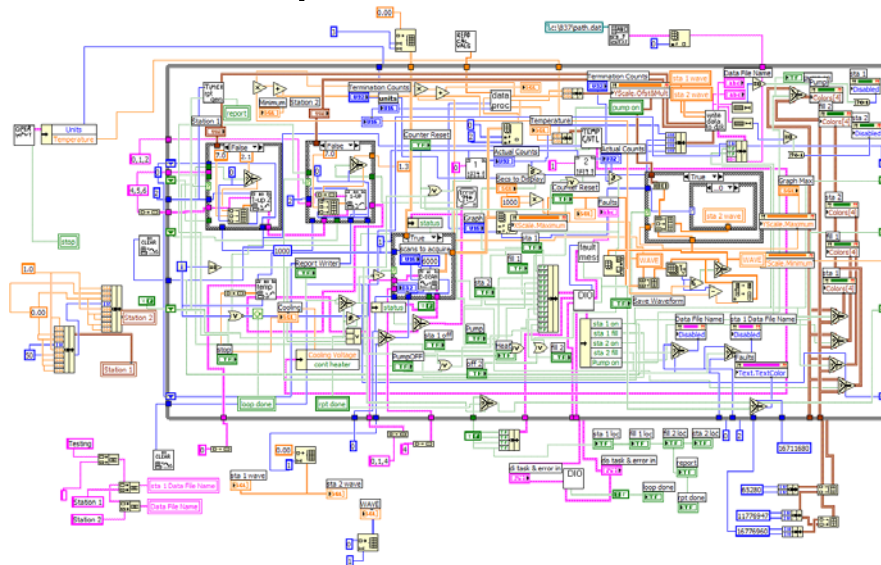
- LabVIEW offers many means to an end
- Programming style is the differentiator
 - Readability
 - Robustness
 - Efficiency
 - Maintainability
- Techniques are style related

ni.com



This presentation was motivated by some common mistakes we've seen made repeatedly by the customers of Bloomy Controls. LabVIEW provides many means to an end. The developer's decisions on how they implement a feature may have a subtle effect on readability, or a profound effect on the overall performance and reliability of the virtual instrument (VI). Hence, LabVIEW programming style is extremely important.

Bad Code Example



ni.com

NATIONAL
INSTRUMENTS

Why is this LabVIEW diagram bad? What would you do differently?

Experienced LabVIEW developers refer to this diagram as “spaghetti.” It exhibits an inappropriate architecture, insufficient modularity, and a haphazard wiring scheme. Specifically, the wiring was performed haphazardly, with data flowing left-to-right, right-to-left, up, down, and all around. Several subVI icons contain text that was drawn free-hand, one unique trait that I would not consider common. Error handling is incomplete, and documentation including comments and VI descriptions, is nonexistent.

When Bloomy Controls was contacted to debug this application, our first request was “please forward us the documentation for our review.” Not too surprisingly, documentation was never written for this application. Even worse, the developer had long since left the company and was no longer available to answer questions.

1. Write a Functional Specification

- Understand the application's requirements
 - Interview operators, developers, engineers, managers, bean counters, etc.
- Formally document the requirements
 - Statement of high-level objectives
 - Specific requirements for I/O, analysis, GUI
 - Assign priorities to each requirement

ni.com



The most common problem I see is that people begin developing their LabVIEW applications without a full understanding of the requirements. This results in poor choices for the top-level architecture, incorrect data structures, inadequate error handling, etc. Moreover, it often results in wasted effort when significant portions of the application must be rewritten to accommodate new definitions of poorly understood requirements. Documenting the specifications, and having them reviewed by all contributors, ensures that the requirements are understood, agreed upon, and approved. This should be the first step in any LabVIEW application development effort, except if you're developing a very simple VI intended only for your own use.

2. Use Proper Data Flow

- Data flow definition

"A block diagram node executes when all its inputs are available. When a node completes execution, it supplies data to its output terminals and passes the data to the next node in the dataflow path."

- Parallel paths are permitted
- Efficiency is optimized when LabVIEW determines the execution order
- Readability is optimized when source terminals, wires, and destination terminals are neatly aligned

ni.com



Data flow is the fundamental principle of LabVIEW. Data flows along wires from source terminals to destination terminals. Programs execute most efficiently when data dependency determines the execution order. Hence, parallel data paths are desirable, unless a specific order is required.

Data Flow Impediments

- Sequence structures
- Local and global variables
- Coercions
- Sloppy wiring

ni.com



Sequence structures force the order of execution where data dependency doesn't exist. Sequence structures are undesirable because they undermine data flow principals and reduce the efficiency of LabVIEW's compiler.

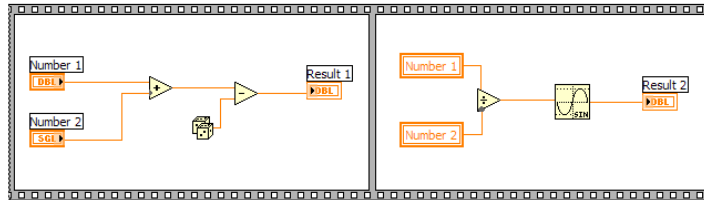
Local and global variables also undermine data flow by duplicating terminals on the diagram. Each Read Local or Global variable makes a copy of the data in memory. Each Write Local or Global variable has the potential to overwrite a value written from another Write operation, resulting in a race condition. Hence, local and global variables are slow to access, consume memory, and add complexity and potential misbehavior. These issues become increasingly important when the data is large or complex, or many variables are used. Experienced LabVIEW programmers generally avoid local and global variables unless absolutely necessary.

Coercions appear as gray dots at the junction of wires and nodes. They indicate that the data type carried by the wire is being converted, or upgraded to a different data type. This adds an extra operation and creates an extra memory buffer. Similar to Read local and global variables, the additional overhead depends on the number of coercions, as well as the size and complexity of the data. Note that in many instances, a coercion is unavoidable, and the extra overhead may be negligible.

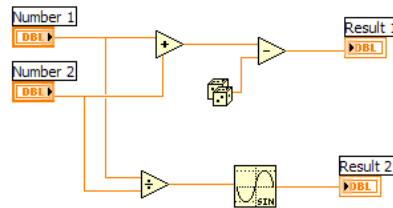
Data flow appears sloppy if there are many unnecessary bends in the wires, or if data flows in multiple directions.

Data Flow Example

- Bad



- Good



ni.com

NATIONAL
INSTRUMENTS

This is an oversimplified example of two equivalent diagrams. The top diagram uses a flat sequence structure to dictate the execution order. The flat sequence is a new feature of LabVIEW 7.0 that shows each frame side-by-side instead of stacked. This improves the readability versus the stacked sequence structures. However, sequence structures are generally undesirable, unless it's critical that the code executes in a specific order and no data dependency exists between the respective nodes.

Notice also that local variables are used to replicate the control terminals for Number 1 and Number 2 in the second frame of the sequence. This is also undesirable because unnecessary local variables increase memory usage, execution time, and complexity.

Finally, the top diagram contains coercion dots because the control storing Number 2 has a different data representation (single precision float) than the others.

The bottom diagram allows LabVIEW to efficiently process parallel data paths.

Data Flow Enhancements

- Artificial data dependency
 - Use error cluster and/or refnum or task ID
- Shift registers
 - Reduce local and global variables
- Clusters
 - Reduce the overall number of wires
- Neat wiring
 - Left-to-right data flow
 - Align nodes to form straight wires
 - Consistent data types
 - Avoid overlapping or obstructions

ni.com



When the order of execution must be specified, an experienced LabVIEW developer will create artificial data dependency using an error cluster or other common data type as an alternative to a sequence structure. Additionally, shift registers can be used to dramatically reduce the required number of local variables. Clusters can be used to combine multiple data elements and thereby reduce the overall number of wires required. Finally, be sure to practice neat wiring practices.

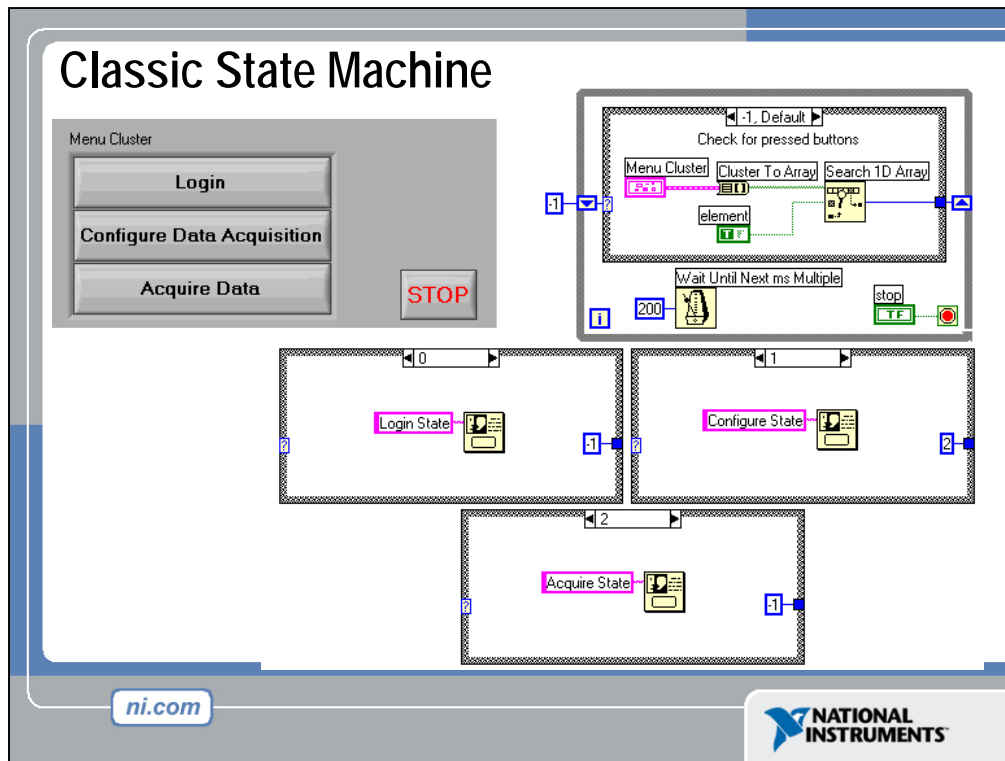
3. Implement a State Machine Architecture

- Define application as a series of states
- Can go to any state from any other state
- Easy to modify, maintain, and debug
- Self-documenting
- Scalable!

ni.com



The diagram of all top-level VIs, except for very simple applications, should contain a state machine architecture. This provides the most flexibility and efficiency, while maintaining the natural data flow of LabVIEW.

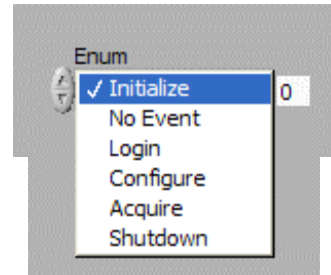


This is a classic state machine, as defined in the LabVIEW Basics hands-on course. It consists of a single case structure within a while loop. A shift register passes the next state as an integer data type wired to the condition terminal. The default case polls a menu of boolean controls. If a button is pressed, Search 1D Array returns the array element (cluster order number) of the first control that matches the True boolean constant, and passes this number to the shift register on the right border of the While loop. Upon the next iteration of the While loop, the state number is read from the shift register on the left border of the while loop, and passes it to the condition terminal of the case structure to select the corresponding frame number. Hence, each button in the menu cluster corresponds to a frame of the case structure, numbered in order of the cluster order of the control within the cluster.

It is important to note that the state machine is not limited in number of states by the number of boolean controls in the menu. Rather, any number of states may be defined and called from any previous state. Furthermore, the state to execute next can be programmatically determined within the previous state. Therefore, state machines provide powerful flexibility.

State Machine Enhancements

- Use enumerated or string for case selector
- Poll user interface events in “No Event, Default” frame or in separate event structure in parallel loop
- Use intuitive state names
- Include “Initialize” and “Shutdown” states

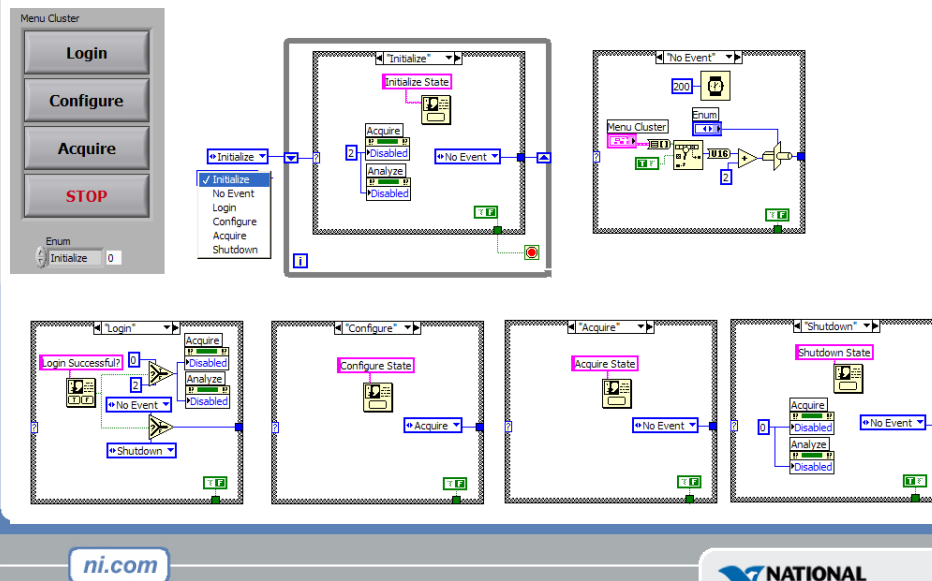


ni.com



An enumerated or string case selector, with intuitive state names, serves to document the state machine. Avoid numeric selector types that display numbers in the case selector instead of text. “Initialize” and “Shutdown” frames are used for functions like initializing or closing instruments and hardware, setting control properties, and reading or writing to configuration files. Using “Initialize” and “Shutdown” states also reduces inefficient block diagram real estate that otherwise would contain these functions outside the state machine.

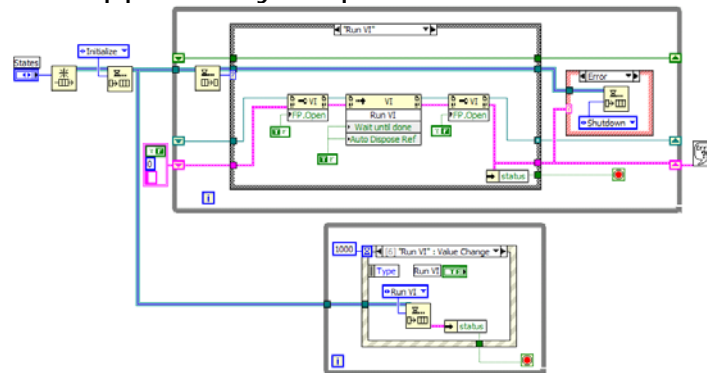
Enumerated State Machine



Here is an enumerated state machine. Notice the intuitive state labels in the case structure selector and in the enumerated constants.

Event-Driven State Machine

- Consider desired response to GUI for determining state granularity
- Consider applicability of queues



States typically are derived directly from the steps of a test, measurement, or control sequence. However, consideration should be given to how long each state takes and how quickly the program needs to respond to GUI events. For example, if your VI needs to execute a Shutdown sequence within two seconds of the user clicking the **Abort** button, then the process should be divided in states with enough granularity such that no state takes more than two seconds. Queues may be used to store multiple user events and pass those events between parallel structures for processing.

The diagram shown above is functionally equivalent to the previous state machine example. However, it uses a separate parallel event structure to optimize its ability to detect user interface events, and it stores the events in a queue for processing by the state machine. It has the advantage in that it can capture and process multiple user interface events.

4. Use Proper Error Handling

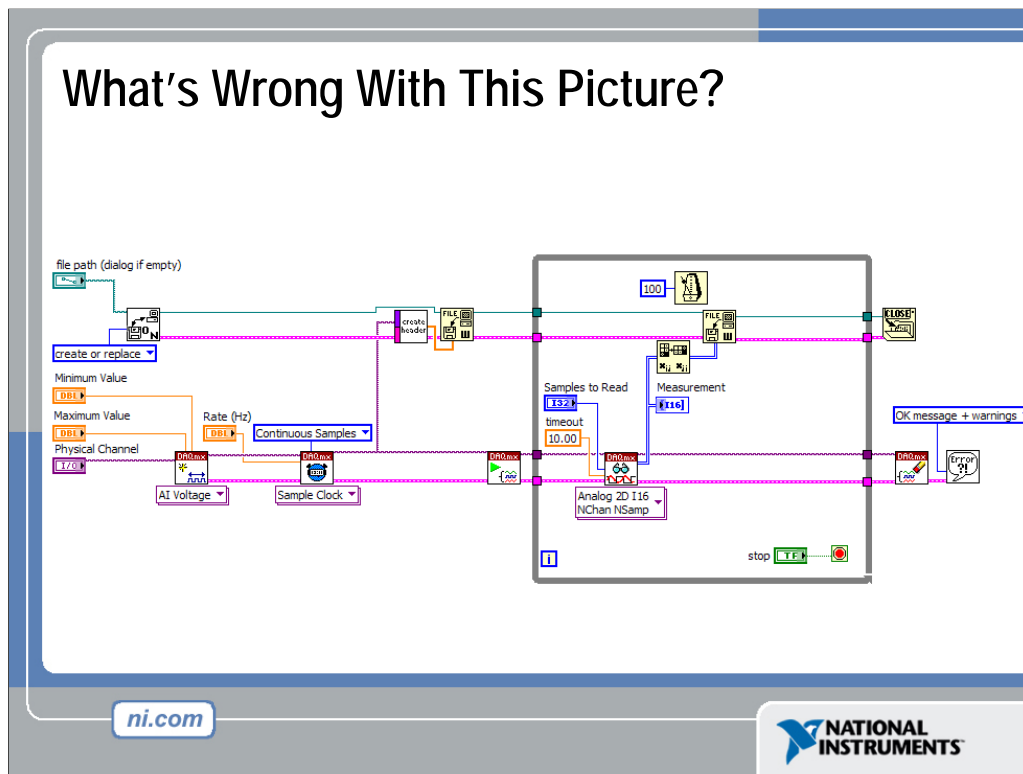
- All VIs must trap and report any I/O-related errors that might occur
 - Trapping is facilitated by propagation of error cluster
 - I/O functions include DAQ, File I/O, Instrument I/O, Communication
 - Reporting methods include dialog prompt or log to file

ni.com



Error handling consists of trapping and reporting any undesirable behavior that may cause the program to malfunction. We primarily are concerned with input/output errors, where LabVIEW is calling a device driver, operating system, or any application or resource external to the LabVIEW environment. Examples include querying an instrument that is not powered on or hung up, or reading or writing to a file or network path that does not exist, and so on. All such I/O functions contain the standard error-in and error-out clusters. These terminals must be wired and propagated among I/O functions and terminated with an appropriate error handler VI.

What's Wrong With This Picture?



The diagram above traps an error only if it is present upon completion of the last iteration of the While Loop. Any errors that are generated by the DAQmx Read VI or the Write File function before the user clicks the **Stop** button will be ignored. Therefore, many errors could go unnoticed!

[illegible]

18

5. Document Your Source Code

- Control labels
 - Use succinct, intuitive labels
 - Indicate units in parentheses or use free labels
 - Enter descriptions or online help where further text is needed
- Icons
 - Intuitive text or graphic
 - 10-point small fonts
 - Color coding for icons of related subVIs

ni.com



There are many techniques for documenting source code. Proper control and indicator labels go a long way toward documenting the front panel and diagram. It is very important to use succinct and intuitive labels, and keep them visible on the diagrams. Enter descriptions for controls and indicators where more information is needed. Icons should properly portray the function performed by the subVI.

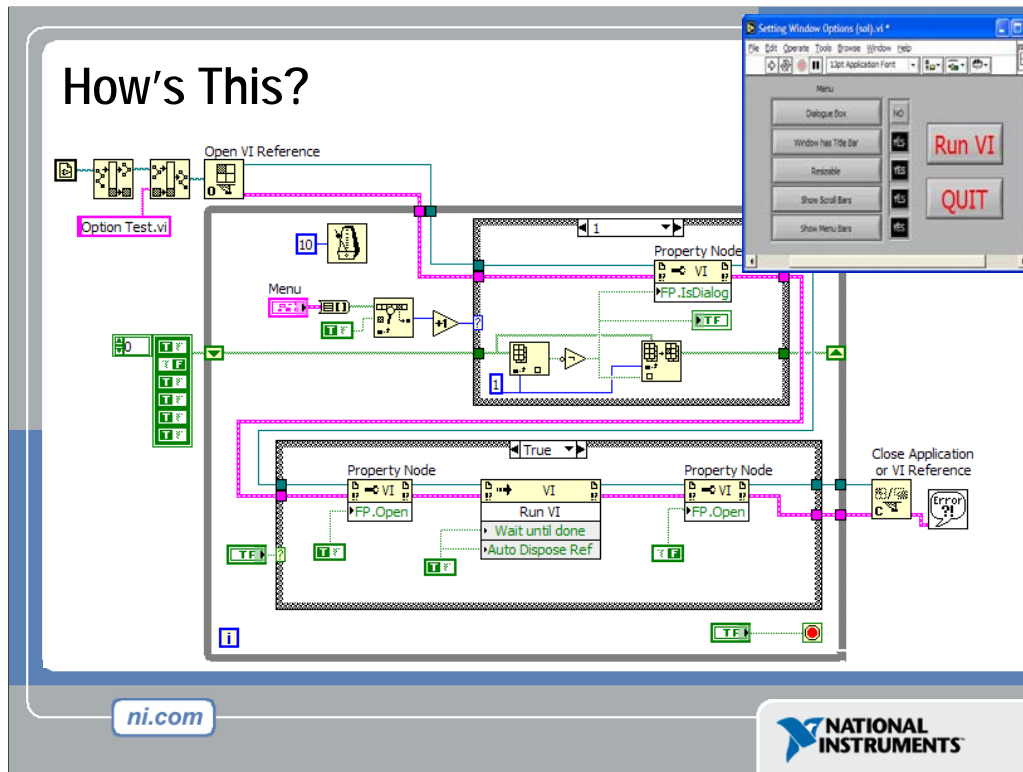
Document Your Source Code (continued)

- Diagram
 - Set all control labels visible
 - Liberally document with free labels
 - Hide subVI labels
- ***Enter descriptions for each subVI!***

ni.com



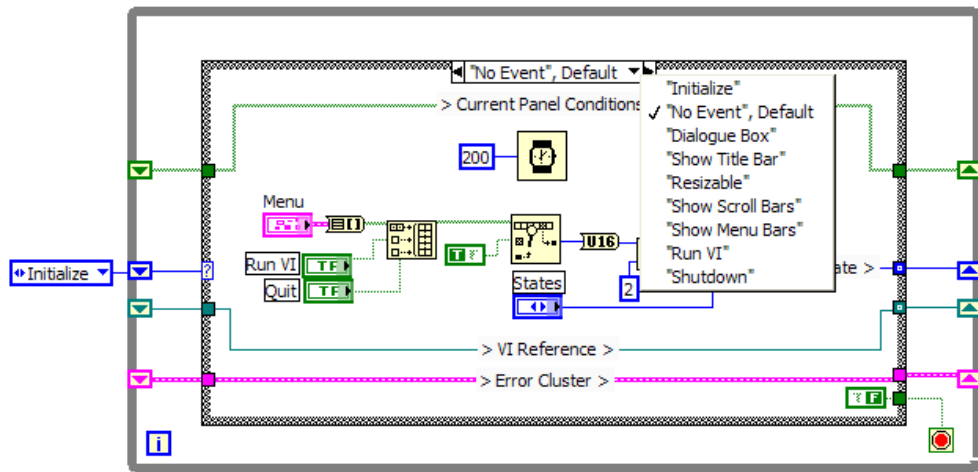
The diagram should be documented liberally with free labels, including long wires and each frame of a structure. SubVI labels tend to occupy valuable space and generally are unnecessary. The most important documentation rule of all is to enter subVI descriptions for every subVI!



Now, let's put it all together by looking at an example. Is this VI a state machine? How's the data flow? Error handling? Documentation?

This example is not a classic state machine because there are two case structures, and a given case cannot affect the next case because the next case is always determined by the control values, instead of passing the next state to the selector terminal via a shift register. The data flows right to left as well as left to right. Control and indicator labels are not visible on the diagram, and comments are absent. Error handling is present, but only errors present during the last iteration of the loop will be trapped. No textual comments exist, except for some of the function labels, which we decided are not necessary because they use up space on the diagram.

Improved Example



This VI's diagram is functionally equivalent to the previous slide. This diagram uses a true state machine architecture with an enumerated selector type and intuitive textual state names in the case structure's selector area. The next state is determined in the previous state and passed to the case's selector through a shift register. Also, several more techniques are applied, including visible owned labels, left-to-right data flow, and free labels applied to long wires. The first error that is generated is maintained in the shift register and reported in the "Shutdown" frame.

Conclusion

- Good LabVIEW programming techniques ensure:
 - You fully understand the requirements before you begin
 - Your software will be neat, robust, efficient, and scalable!

Contact Us

Peter Blume
President/Chief Engineer
Bloomy Controls Inc.
Windsor, CT
(860) 298-9925
peter.blume@bloomy.com
www.bloomy.com



ni.com



One final note ... Due to time and space limitations, we are not including the conforming solutions to the nonconforming applications discussed at the beginning. These and other slides, however, are posted and available for free download from **www.bloomy.com**.