Designing a Large GUI Application in LabVIEW

Damien Gray

16 August 2004

Overview of the Problem

Designing a large GUI-based application in LabVIEW is not much different from any other programming language. The key element is planning. Small LabVIEW applications usually do not require much planning, since programming in LabVIEW is essentially like drawing a flow chart. This lulls the casual user into the false assumption that they can do the same for large programs. The single most common reason for a large LabVIEW application to fail is that is was grown from smaller applications and the final program structure is inadequate for the task. Debugging and maintenance become a nightmare until the program structure is corrected.

So how do you avoid this problem? First, look at what you are trying to accomplish. This is a GUI based application, so the GUI should dominate the planning. Plan the GUI first, based on what it is supposed to do. LabVIEW makes it very easy to mock up GUIs in very little time. Try several alternates. Run them by your co-workers. Make sure you use established norms. For example, don't use Ctrl-T to exit the application under Windows. That should be Alt-F4 or Alt-F-X.

This brings up another very important point. This is a large GUI based application. If it is done well, there will be several ways to do almost anything. To continue the previous Windows example, to exit the program, the user may press Alt-F4 (system program close), Alt-F-X (keyboard shortcut for menu File->Exit), select the menu item File->Exit, or click the top right exit button. Your program needs to handle all these methods through the same code, or you will end up with maintenance problems. Two methods help to solve this problem – modularization and proper program structure.

Modularization

Modularization is breaking your code into modules of functionality and data which relate to one another. The easiest way to do this is to use object-oriented design techniques. If you do not know object-oriented design, learn it. It will save you endless hours of frustration. LabVIEW, like C, does not enforce an object-oriented approach. However, it can be used in a pseudo-object-oriented manner, leading to easier maintenance and modification of the code.

Proper program structure is the key to object-oriented design (as well as other things). A module or object is usually composed of data and functions. You can have data objects and function objects, but an object usually contains both. For example, an oscilloscope application will have objects for the vertical and horizontal

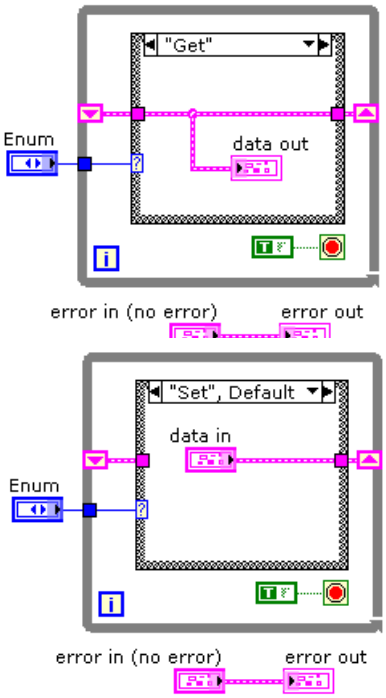
controls. The objects will contain the current settings and functions to change and access these settings. This is a key point. In object oriented programming, the programmer does not usually directly read a piece of data. It is changed and accessed through functions. This allows the programmer to restrict data access to avoid race conditions and to execute conditional code when a variable is set. An example is in order.

Suppose you have that oscilloscope application and you use LabVIEW globals to store your data. At point 1 in your code, you read the horizontal scan rate, set the oscilloscope device to it, read back the actual rate, and reset the horizontal scan rate to the one which was actually set by the device. At point 2 in your code, you read the horizontal scan rate, convert to a time per record, query the user for a new time per record, and write the new horizontal scan rate. You then set the scope to the new scan rate. However, you have a race condition. If the point 2 code reads the scan rate, then the point1 code reads the scan rate, then the point 2 code writes the scan rate, then the point 1 code writes the scan rate, the scan rate the user specified will be overwritten and not executed. You need a way to prevent this common type of race condition. This brings us to the topic of program structure.

Program Structure

Functional Global Databases

Uninitialized shift registers in FOR or WHILE loops can be used to hold data. As long as they never go out of memory, they will hold the last state of the shift register. This method is referred to as a LabVIEW 2 style global or a functional global. The advantage of this method over the usual LabVIEW global is that access to the data in the shift register can be controlled as much or as little as desired. The code at right shows a simple functional global with set and get functions. The data input is a strict typedef cluster for ease of maintenance.



This example does not implement data locking, but this is fairly simple to do using semaphores. Save the semaphore reference in the database and use it to restrict access to the database when you want it to. At the very least, you need to lock the database when it is being read preparatory to being written. This prevents changes in the database between the time the data is read and the time it is processed and rewritten. The beauty of this technique is that it can be customized to your application. You are only limited by your programming skills. You can include multiple instances of the data (the shift register is an array) with or without reference counting. You can lock the data for both reads and writes.

You will need methods to properly initialize the data. In C++ terms, these are the creation and destruction methods. You will also need to write accessor VIs (read and write) for every variable in the data cluster. While this may seem overkill, it pays off every time you need to put a restriction or condition on a variable access. You do not have to go searching through your code for every call which looks at the particular variable. All you have to change is the accessor, and you are done. Using the oscilloscope example from above, you have written the application, but now find out the oscilloscope does not support 20V/div while in 50Ω input impedance mode. All you have to do is change the impedance accessor so the V/div is coerced down if it is 20V/div and the user tries to set 50Ω as the input impedance. This is much easier than trying to fix this at every point in your code that impedance is set (there are several, remember).

Event Handlers, Task Handlers, and State Machines

Since there are several ways to do almost everything, you need an easy way to only write the code once for any task, but get to it from many places. The solution is to separate the two processes. At minimum, you need an event handler and a task handler. The event handler is very easy. Use the LabVIEW event structure inside a WHILE loop. With LabVIEW 7.0 and higher, you can define your own events, so you can trap almost any event with it. In addition, you can programmatically send events to the event handler either by firing user events or changing front panel controls with the `Value (Signaling)` property. A common new user mistake is to use the event handler as your task handler. Do not do this. It is difficult to maintain, since you have many events which lead to the same functionality. This leads to multiple edit locations for simple changes in functionality.

The easiest way to create a task handler is the queue-loop-case statement method. Create a queue which will function as your task list. The queue data type should be either a strict typedef enum or text. This choice depends upon the application and the programmer. Both work well. Text is easier to add elements to and can have dynamic elements. The enum gives easier coding because wiring it into a case statement guarantees the cases will be correctly set up. However, the enum cannot be programmatically expanded to include more cases.

After you have your queue, run this into a WHILE loop. The first element of the while loop is a dequeue VI with infinite timeout. When it receives an element, the element is used as a switch for a case statement, which is the task. This gives you a task handler which does not consume CPU time, except when it is actually doing a task. The event handler also has low CPU overhead, so you get a low overhead mechanism for user interaction.

What about states? You can implement a state machine on top of this in two basic ways. You can create a standard state machine and populate each state with a different task handler. The queue is still the same, but the tasks are handled differently. Alternately, you can put a case statement in each task, modifying the tasks based on the state of the program. Either work well and it is a matter of taste

as to which you pick. Note that a state machine is not necessary for all GUI programs. Many only have one state – run.

Tips and Tricks

Disk Layout

Now that you have converted your design to an object-oriented approach, how can you easily keep it there? Since LabVIEW does not have a project window, you can use the file browser for one, if you exercise some discipline. Use a single folder for the project. Use subfolders for each object. In each object, use a folder for the data (any typedefs) and a folder for the methods (all the VIs – databases, accessors, creation, destruction, etc.). By doing this, you will be able to fairly quickly find any piece you are looking for – even two years after you wrote it.

The names of everything should contain three elements – the project, the module, and the function. The project and module should be short acronyms at the beginning of the name. This ensures both ease of use and a unique name for the VIs. Continuing our oscilloscope example, the name for a VI which sets the vertical range could be OscpVert_SetRange.vi. Be sure to watch name size limits on the platform you are targeting.

Documentation

As you write your code, document it. This is critical for maintenance. At the very least, every control on the front panel of every VI should have the documentation filled in. Ideally, the VI property documentation should also be filled in, but with most accessors, this is not necessary. If you just completed a fairly tricky piece of code, put a comment on the block diagram to explain why you did things the way you did. Document bug workarounds in the same fashion so they can be removed at a later time. Comments are best kept near the code they refer to. If you are tight on space, you can use numbers near the code which refer to a numbered list below. Put a light yellow background on your comments if you need them to stand out. Use a light orange background on bug workarounds to make them distinct.

Icons are also necessary for VIs. Icons should have the same information as the file name – project, group, and function. This can be as simple as a light blue background means the project and an up arrow on the border means the vertical setting module. Unless you are doing code which will be released on the palette to customers, text is fine. If you want to make nice icons (gradient fills, reusable glyphs, etc.), use a real bitmap editor (e.g. GIMP, PaintShop Pro, Photoshop in order of price). The LabVIEW editor is not suited for serious icon design. GIMP is free.

Debugging

The best thing you can do for debugging is to keep your diagram clean. Do this as you work, not at the end. By the time you have finished writing a VI, 80% of the debugging is finished and the reason for keeping the code clean is correspondingly less. It will take you longer to initially write code, but will take less time to get it out the door. The act of cleaning your code as you write it forces you to think about it

and results in many fewer minor bugs, such as wires connected to the wrong terminal.

By the same logic, a VI is not finished until it has an icon. Icons don't do you any good when the code is finished. They are only useful while you are working on it. Putting an icon on a VI lets you know what it is when you are using it. Waiting until you are finished with a project to add icons helps you debug future problems and all but ensures that you will have them.

Large GUI problems are often race conditions. Finding these requires knowing what your data is at any given step of the program. The use of the functional global database makes this fairly easy to accomplish. Since the database can be read any time, anywhere (unless you lock it to reading), you can create a data browser which allows you to look at your current data in almost real time. This is a must. All databases should have this browser.

Refactoring

Refactoring refers to rewriting portions of your code to make it more efficient or to solve serious design issues. In addition, LabVIEW constantly introduces new capabilities which make it easier to produce better and more maintainable code. Using these can sometimes result in major benefits, but require major changes. For example, the event structure introduced in LabVIEW 6.1 introduced huge efficiency gains for GUI applications, but required major rewrites to take advantage of it. Do not be afraid of refactoring. It keeps your code efficient and prevents the infinite patch syndrome (patch a problem, patch the patch, repeat until fired). However, beware of large refactoring efforts close to release. Unless you absolutely cannot put it off, delay these until the start of the next revision of a project. A major project revision will almost always include some major refactoring. This should be done first to weed out all the bugs by release time. Good software is evolutionary. Use this to your advantage.

When you write a new program, expect to do a major refactor before the first release. Typically, the program should be written and user tested, then the GUI modified and the code refactored before a second round of testing. Do not be afraid to throw away code. It is rare that you get everything right the first time. Your source code control allows you to easily restore your previous code, if necessary.

Source Code Control

Use it. It helps you in several ways. It provides a backup in case your development machine dies. It provides convenient restore points, should you have an editing session which goes really bad. It facilitates large, multi-person projects by taking care of the bookkeeping of who has what at what time. LabVIEW 7 and higher integrates fairly well with most standard source code control modules. However, source code control systems treat LabVIEW code as binary, so cannot save just differences. As a result, make sure you only check in things that change, or your source code control system can easily run out of disk space.

Synopsis

- Plan
- Design GUI around the problem
- Use object oriented techniques to modularize your code
- Keep your code clean
- Use proven program structures
- Document your code as you write it
- Rewrite code any time it needs it
- Use source code control

Resources

The following resources can help you in your large GUI project development.

- **DevZone**
- **OpenG Project** (http://www.openg.org) - web site dedicated to expanding LabVIEW in an open source manner
- **LabVIEW Technical Resource** (http://www.ltrpub.com/) – magazine dedicated to LabVIEW
- **LabVIEW Web Ring** (http://www.webring.org/cgi-bin/webring?ring=labview;list) – webring dedicated to LabVIEW. Links and data are sometimes a bit old.
- **Google**