An Easy-To-Maintain Configuration File Architecture

A LabVIEW Architecture Solution for the Dynamic Environment

by David Thomson

When developing new experiments and instruments at the National Oceanic and Atmospheric Administration, it is often the case that LabVIEW software must evolve with the instrument. In the ideal case, software is thoroughly designed before the first front panel is created. During instrument development, it is often impossible to follow this ideal model. As the instrument evolves, software is needed to control and acquire data from an ever-changing assembly of components. These components often include stand-alone instruments as well as data acquisition devices, all of which need configuration information associated with them. This configuration information usually includes device numbers and addresses, channel names and parameters, experiment descriptions, data file paths, and so forth. A significant effort in developing such software is often the creation of a mechanism for managing this configuration information. We present here a configuration file architecture that is tailored to this type of dynamically-changing development situation. In creating this architecture, we paid particular attention to ease-of-maintenance and clarity of the code. Our architecture is included on this issue's LTR Resource CD.

We present here a configuration file architecture that is tailored to the dynamically-changing development situation. In creating this architecture, we paid particular attention to ease-of-maintenance and clarity of the code.

There are a number of approaches one might take when faced with the problem of saving configuration information to a file. In the user interface, it is typical to group configuration parameters into clusters of controls, with each cluster corresponding to a device or instrument, and the virtual controls often mimicking the appearance of similar real controls on the hardware. In previous projects, we sometimes created configuration files that used binary files to write these clusters directly to disk. Although the file reading and writing is quite simple initially for this method, the situation quickly becomes difficult to manage when the experiment evolves and new parameters must be added to the configuration. Each time a new control or cluster is added to the main configuration cluster, the previously used configuration files become unreadable. By embedding version information in the cluster, it is possible to create "smart" file readers that can read old configuration file versions, but maintenance of such code is time-consuming and prone to error.

Text-Based Configuration Files

When National Instruments introduced configuration file VIs, a different approach became possible that had several key benefits. With these VIs, you can create human-readable configuration files. In addition, version compatibility became much less of a problem. When new parameters are added to a system, old configuration files can still be read, and default values are supplied for the new parameters that are not found in the file. If existing parameters are deleted, their continued presence in the configuration files is of little concern, and they are ignored when the desired parameters are read. This version-independence is the key feature of the Configuration File VIs that led us to adopt them for our type of continually changing experiment development projects.

Although the text-based .ini files have these key advantages, the typical implementation demonstrates several distinct disadvantages. One disadvantage becomes apparent when first looking at the example Read Configuration Settings File.vi, supplied by National Instruments. To find this VI, select *Help* >> *Find Examples*. Choose *Browse According to Directory Structure* and select *File* >> config.llb. This VI demonstrates how five parameters are read from a file. Because the parameters are not in a cluster, they would require five connections on a connector pane. In our experience, the number of parameters required for a fairly complex experiment will quickly exceed the capacity of the connector panes. The obvious solution to this is to bundle the parameters into clusters. This is a simple problem to remedy in the example. Our architecture will expand on this to present a cluster structure that is applicable to many systems and that has additional properties that allow it to work well within our program.

A second disadvantage of the typical .ini file implementation is that the Read and Write functions are generally contained in two separate VIs, as is the case for the NI example. This means

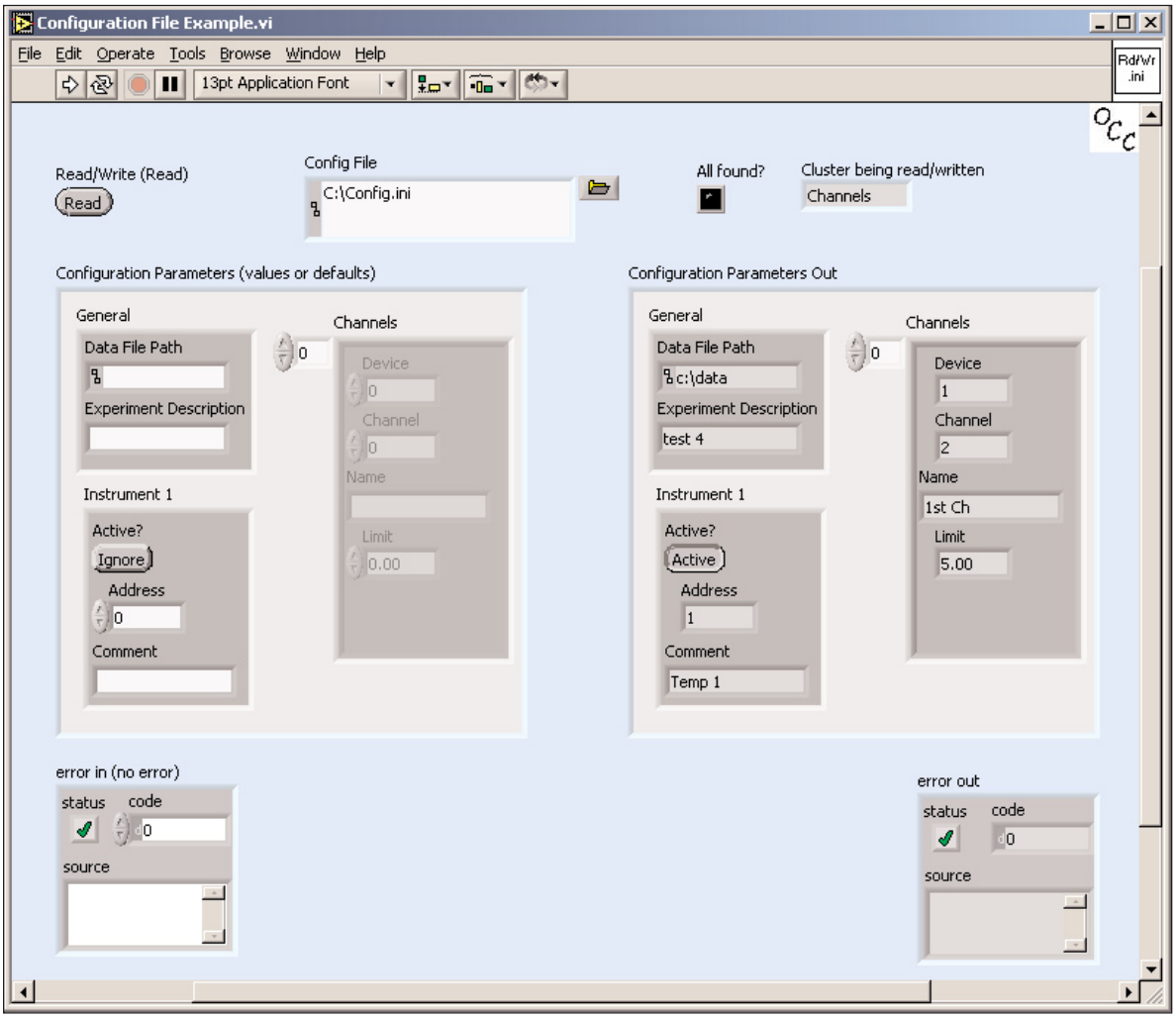

Figure 1: Front Panel of Configuration File Example.vi

that as parameters are added and changed, both of these VIs must be updated and kept in sync with each other. Although this may sound fairly trivial, experience has shown that this process is a significant source of potential bugs. It is far too easy to misspell a parameter name in one of the two VIs, with the result often difficult to detect, let alone track down.

An Improved Configuration File Architecture

Figure 1 illustrates the front panel of the Configuration File Example.vi, showing the input and output clusters and the Read/Write control. *Figure 2* illustrates the block diagram and shows the Case Structure (with one case for each subcluster) inside the For Loop, which cycles through all the subclusters. (The VIs were developed in LabVIEW 6.1. Versions for 6.0 and 6.1 are available on this issue's LTR Resource CD. Before running the examples, set the configuration file path name to an appropriate value for your system.)

The parameters to be read or written are defined in the main cluster, which is a type definition and appears twice on the front panel, once as an input and once as an output. This cluster is made up of subclusters, each of which represents one instrument or logical group of parameters. Although more complex structures could be implemented, this example supports two types of subclusters: flat clusters containing any combination of simple data types, and arrays of similar flat clusters. Although a single cluster of parameters can get rather large for complex experiments, it does provide a convenient way to wire the parameters throughout the program.

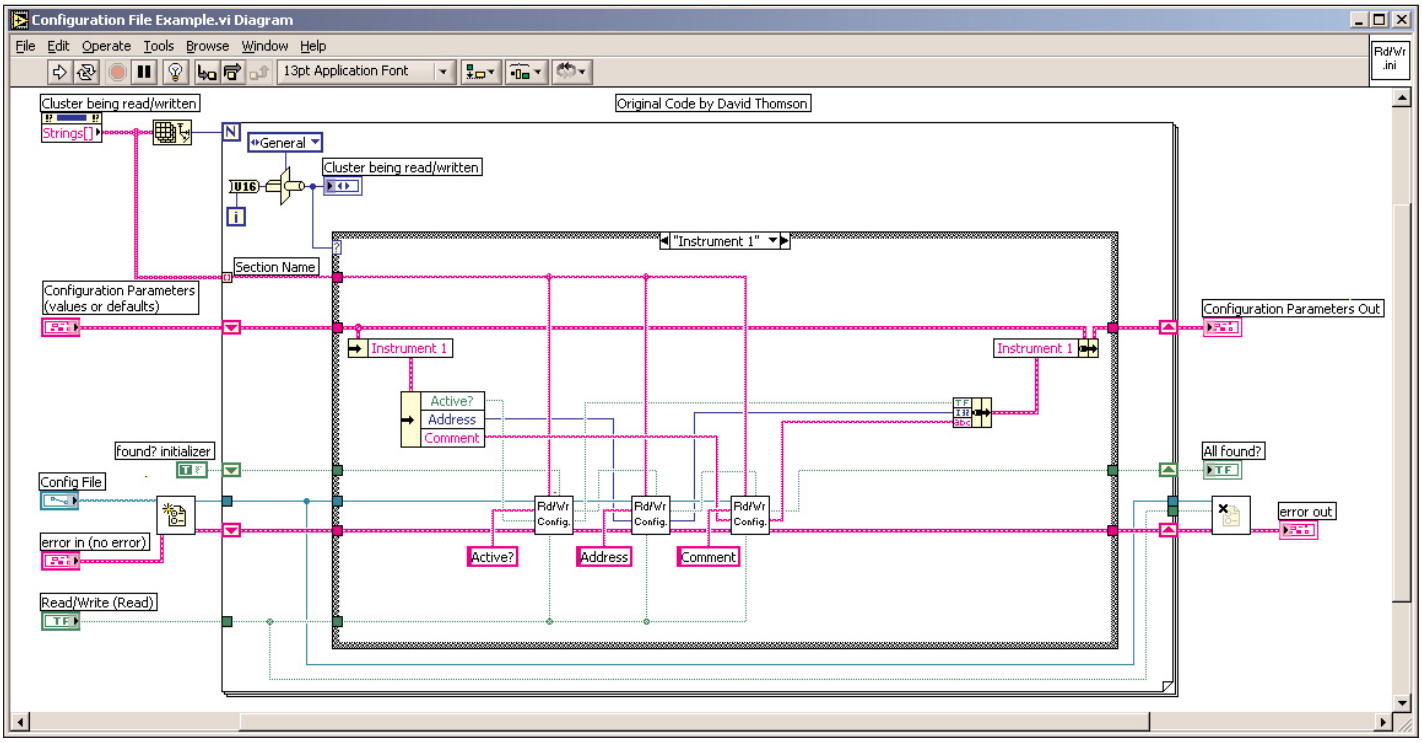

Figure 2: Configuration File Example.vi Block Diagram

If the main cluster becomes too unwieldy, it can be split into several distinct clusters, each one using a separate implementation of the architecture shown here. These distinct parameter clusters can be stored in the same .ini file, even though different VIs at different places in the software read and write to them.

Another key feature of this architecture is evidenced by the *Read/Write* control in the upper left corner of ***Figure 1***. As this control implies, this single VI handles both the reading and the writing of the parameters from and to the .ini file. When reading parameters, the input cluster can be used to provide default values, and the resulting values read from the file are in the output cluster. When writing parameters, the values to be written are put in the input cluster, and the output cluster is a redundant copy of those values. By combining the read and write functions into one VI, the maintenance of this architecture is reduced by at least a factor of two. When new parameters are added, only one VI must change. This automatically eliminates the potential for misspelling the parameter names in the Read and Write functions, as described previously.

To simplify the programming of a single VI that both reads and writes parameters, we created a version of the Configuration File VIs that handles both these functions called Read_Write Config Data.vi. This VI is polymorphic, as are the original Configuration File VIs, to further simplify the process of adding and altering parameters. This VI can be seen on the diagram in ***Figure 2***. It has the same inputs and outputs as the Configuration File VIs, with the addition of the *Read/Write* control and another control called *Previous Found?*. The *Previous Found?* input is a mechanism for connecting the found? outputs of the Configuration File VIs so that no additional logic is needed on the main diagram to determine whether all the requested parameters were found in the file. If *Previous Found?* is left unwired, the found? output acts as it does in the original Configuration File VIs, indicating whether the requested parameter was found. If however, the *Previous Found?* input is wired from the *found?* output of the previous Read_Write Config Data.vi, it automatically acts as an accumulating indicator for all the parameters that are wired together in this way.

Program Structure

As shown in ***Figure 2***, the structure of the example program is a Case Structure inside a For Loop. The For Loop iterates once for each subcluster. The Case Structure has one case for each subcluster. The selection is wired from the for loop index and converted to a *Clusters* type definition enumerated constant.

The *Clusters* type definition provides several maintenance advantages. This type definition must be edited manually to contain one entry for each of the subclusters of the main parameters cluster. Although this editing adds one more step to the maintenance, it results in other savings and features. By sizing the array of the *Strings* property of this type definition, the number of For Loop iterations is calculated automatically, eliminating the need to manually change a diagram constant. Furthermore, the type definition automatically documents the Case Structure by providing the subcluster names as the case names. By not having a default case, the type definition provides further insurance against maintenance errors by forcing the Case Structure to be updated whenever a new subcluster is defined. Finally, the *Strings* property of the type definition is used to automatically supply the section names for the configuration file.

Within each case, the relevant subcluster is unbundled by name, then its parameters are unbundled by name, the reading/writing is done, the values are rebundled, and finally the subcluster is rebundled by name. Using the standard bundle (rather than bundle by name) for the first bundling provides another mechanism for preventing maintenance errors. When a new parameter is added to a subcluster, this bundle will no longer match the required structure of the second bundling, causing a broken wire to appear, and forcing the diagram to be updated to include the new parameter. This is desirable because overlooking the new parameter update will result in an old value being passed to the output cluster. Readability is still maintained, however, by having the unbundle by name on the left side of the case, so that each parameter is automatically documented.

Besides flat clusters of simple parameters, it is often useful to store arrays of clusters, such as might be used to configure multiple similar channels of a device. The third case (*Channels*) demonstrates how this can be accomplished. (You are encouraged to review these VIs, which are included on this issue's LTR Resource CD.) An additional For Loop indexes the array and the section name is automatically created from combining the array name with the For Loop index. There is one significant issue to keep in mind when using this type of array of clusters: how to handle the default values of the array elements. For each application, the programmer should decide whether the number of array elements to be read should be determined by the number of default values supplied, by the number of array elements already in the file, by the larger of these two numbers, or by some other criteria. In this example, the number of array elements written to the file is stored as another parameter in the file. When the array is read from the file, at least as many elements will be read as currently exist in the file. This handling of array elements should be carefully considered when these types of configuration files are used.

In the example presented, the names for each parameter are wired as constants on the diagram. It is fairly simple to automate this aspect as well, to eliminate programming errors. The program Configuration File Example Auto Names.vi includes a subVI that parses the parameter names out of the type definition cluster. These names are supplied in a 2D array, which is then indexed by the main For Loop, and then by an Index Array function, to supply the names to the Read_ Write Config Data.vi. Although this further automates the maintenance of the program, it provides additional complications when more complex data structures are used. The parsing subVI was written to handle only flat clusters and arrays of flat clusters. If another data structure is used (such as a cluster of clusters), the parsing VI would have to be updated.

The architecture presented overcomes several difficulties and limitations associated with other methods of maintaining configuration files. We have found it to be useful in accelerating the development of configuration file handling and in reducing the number of bugs and errors generated as these configuration file handling VIs are maintained.

LTR Publishing would like to thank Wilbur Shen of G Systems for his valuable technical assistance with this article.

About the author:

David Thomson is a Research Scientist and Certified LabVIEW Developer at the NOAA Aeronomy Lab, where he has been using LabVIEW for the last 10 years in the development of airborne atmospheric chemistry instrumentation. For the past 4 years, he's been the principle of Original Code Consulting (www.originalcode.com), an NI Alliance member.

LabVIEW Technical Resource is an independently produced publication of LTR Publishing, Inc.

LabVIEW is a registered trademark of National Instruments Corporation.