

Error Handling and its Implementation in LabVIEW

Robust computer programs must detect errors and recover from them where possible. In this article, I'll discuss error handling in general, the standard LabVIEW technique for implementing it, and some tools for making the implementation easier. If you're already familiar with the basics, you may want to skip ahead to the more advanced topics.

Error Handling

Computer programs are always subject to errors, either because of programming mistakes, invalid runtime inputs, or unavailability of required resources. Error handling is critical to coping with these sources of errors in a user-friendly fashion. Programs with no error handling will crash or invisibly perform incorrectly when an error occurs. Programs with primitive error handling often display cryptic error messages and then exit. A program with sophisticated error handling will continue to run, automatically fix problems if possible, and if not, switch to a safe state appropriate to the error and inform the user.

The first step in handling errors is to immunize your code to as many of them as possible. For example, numeric controls on front panels should be set to coerce all entries into an acceptable range. The next step is to determine where in your code you can't avoid the possibility of invalid values. At each location, you then need to determine what kind of response is required, and therefore how the error handling should be implemented.

Robust programs have many error checks, so it's possible for all the error handling code to obscure the code for normal operation. That's why some programming languages have dedicated mechanisms for separating out the error handling code, leaving just the error test in the main line. There's no such mechanism in LabVIEW, but the standard error scheme in LabVIEW provides some of the same benefits.

Standard Error Scheme in LabVIEW

Most of the VIs that are subject to errors in LabVIEW have input and output error clusters. These VIs are designed to be used in chains like the one shown in Figure 1. The last VI in the chain, *Simple Error Handler.vi*, is responsible for displaying error messages to the user based on the error cluster it receives. The other VIs must check their input error clusters to see if a preceding VI has reported an error. If there's an input error, a VI can skip its actions (except deallocation) and must pass through the error cluster unchanged. If there's not an input error and a VI performs its actions successfully, then it passes through the "no error" cluster. When a VI detects a problem, it must construct an error cluster from a True boolean, an error code, and a string indicating the source of the error.

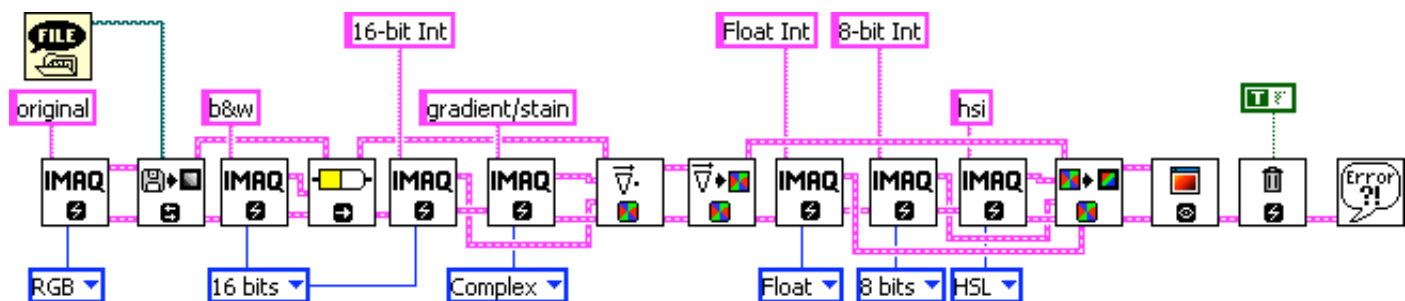


Figure 1. This image processing program illustrates LabVIEW standard error handling. The 7th, 8th, and 12th subVIs have been written to integrate easily with the other VIs, all from National Instruments. An error at any stage will skip the remaining processing and be reported by the error handler.

The block diagram in Figure 1 is unusual in that every link in the chain is a subVI with standard error handling. In general, you'll need to insert case structures into the chain to skip over functions that don't use

error clusters. This is illustrated in Figure 2, where a potentially time-consuming call to *Complex FFT.vi* is skipped if earlier file operations have failed. Since the case structure contains an explicit implementation of the standard error scheme, the chain architecture can continue after it.

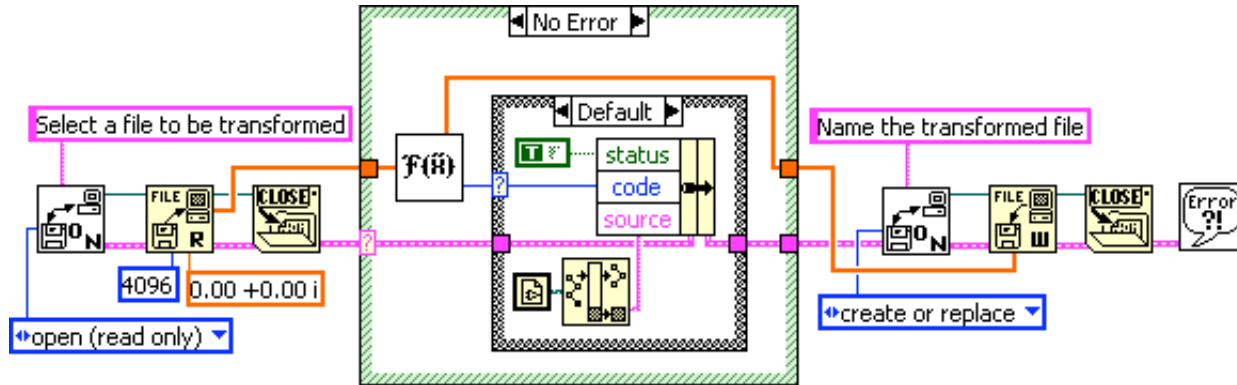


Figure 2. Case structures must be included in an execution chain in order to control functions that don't have error cluster inputs and outputs.

By adopting the standard scheme, you'll find that your code can handle errors efficiently and integrate tightly with the LabVIEW distribution. But to use the scheme, you'll need to build your own error clusters, figure out what error codes to use, and make proper use of the error handler VIs. The enhancement VIs provided with this article assist with all three jobs.

Building Error Clusters with Reserved Codes

You can manually assemble an error cluster whenever needed as in Figure 2, but you'll end up with lots of redundant code on your diagrams. A slightly easier alternative is to wire the optional inputs of *Simple Error Handler.vi* plus set the type of dialog to "no dialog." Since *Simple Error Handler.vi* follows the error handling scheme described in the previous section, it automatically determines whether to pass through its error cluster input or construct a new cluster from the values you specify. Easier still is to use one of the error insertion VIs provided here, for example *Insert Reserved Error.vi* shown in Figure 3.

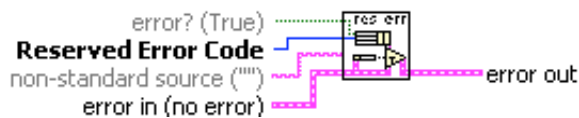


Figure 3. This VI inserts error codes 0-107 which are defined by LabVIEW and are listed in the Error Ring constant.

Each error insertion VI includes an error cluster input, the logic to determine when to pass the error cluster through, and some additional features:

1) "error?" input

The optional Boolean input "error?" can be wired with the result of a test.

2) "non-standard source" input

The error source string input "non-standard source" provides an optional input field for specifying the source of the error. If this input is unwired, the error insertion VIs automatically build a source string based on the "calling chain" – this includes all the VIs in the subVI hierarchy from the VI that calls Insert Error (i.e. the point where the error was detected) all the way up to the top level VI. In cases where the calling chain is insufficient to determine the location of the error, use this input to provide more useful diagnostic information.

3) Error code input

The error code input uses a text ring or enumeration input that's specific to the type of error being inserted. A constant created for this input will be self-documenting because it will appear on the diagram as an error name rather than an error number.

Insert Reserved Error.vi (Figure 3) is used for error codes in the range 1-90, the "G Function Error Codes." The meanings of error codes in this range are defined by LabVIEW, but your VIs can use the existing definitions. The "Reserved Error Code" input is wired with an Error Ring constant, which you can create by popping up on the input terminal or by selecting it from the Additional Numeric Constants palette.

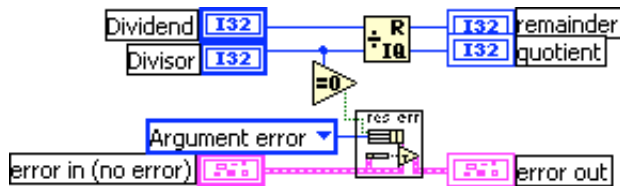


Figure 4. This example shows the use of *Insert Reserved Error.vi*. It adds error reporting of a divide by zero condition to the *Integer Divide* function.

A simple example of the use of *Insert Reserved Error.vi* is shown in Figure 4. *Integer divide+.vi* adds a check for division by zero when using the *Quotient & Remainder* function with single integer arguments. Because integer division of scalars is fast, there's no need to skip the operation in the presence of an incoming error. *Insert Reserved Error.vi* determines if the incoming error cluster is passed through or if a new cluster must be constructed. The error code constant is *Argument Error*, which has a numeric value of 1 and for which *Simple Error Handler.vi* will display a dialog with the message "An input parameter is invalid."

Building Error Clusters with User-Defined Codes

What if you want to use error messages that aren't in the reserved error code list? You will need to adopt user-defined error codes and messages, for which LabVIEW allocates codes in the range of 5000-9999. LabVIEW also provides the *General Error Handler.vi* which has inputs for an array of user-defined codes and an array of corresponding messages. The enhancement VIs provided here are templates for inserting user-defined errors, constructing arrays of codes and messages, and calling *General Error Handler.vi* efficiently.

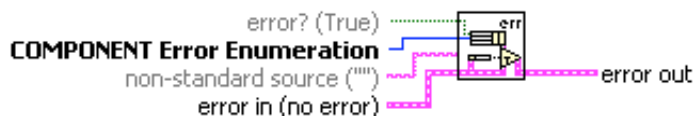


Figure 5. This is a template VI for handling user-defined error codes. It doesn't use the standard LabVIEW template mechanism because it is cross-linked with other parts of the template.

COMPONENT Insert Error.vi (Figure 5) is the template for error insertion. Each instance that you create will be able to insert codes in a contiguous range. If you build a project from independent software components, each component can have its own error code range. This is a good model for projects built from multiple instruments, each with its own LabVIEW drivers. If the ranges of components conflict, the base error code for a component can be changed in a single location to eliminate the conflict.

COMPONENT Insert Error.vi is just like *Insert Reserved Error.vi*, except that the "Reserved Error Code" input has been replaced by "COMPONENT Error Enumeration." "COMPONENT Error Enumeration" is based on a type definition, *COMPONENT Error Enumeration.ctl*, which you edit in order to define your error messages. On the block diagram of the LabVIEW 5.1 version of *COMPONENT Insert Error.vi* (Figure 6) is a labeled constant, "COMPONENT Base Error Code," which determines the starting number for the error codes. NOTE: The LabVIEW 6.0 version of *COMPONENT Insert Error.vi* (not shown) uses the

default value of a front panel control as the base error code which allows the base to be set programmatically when the template is instantiated.

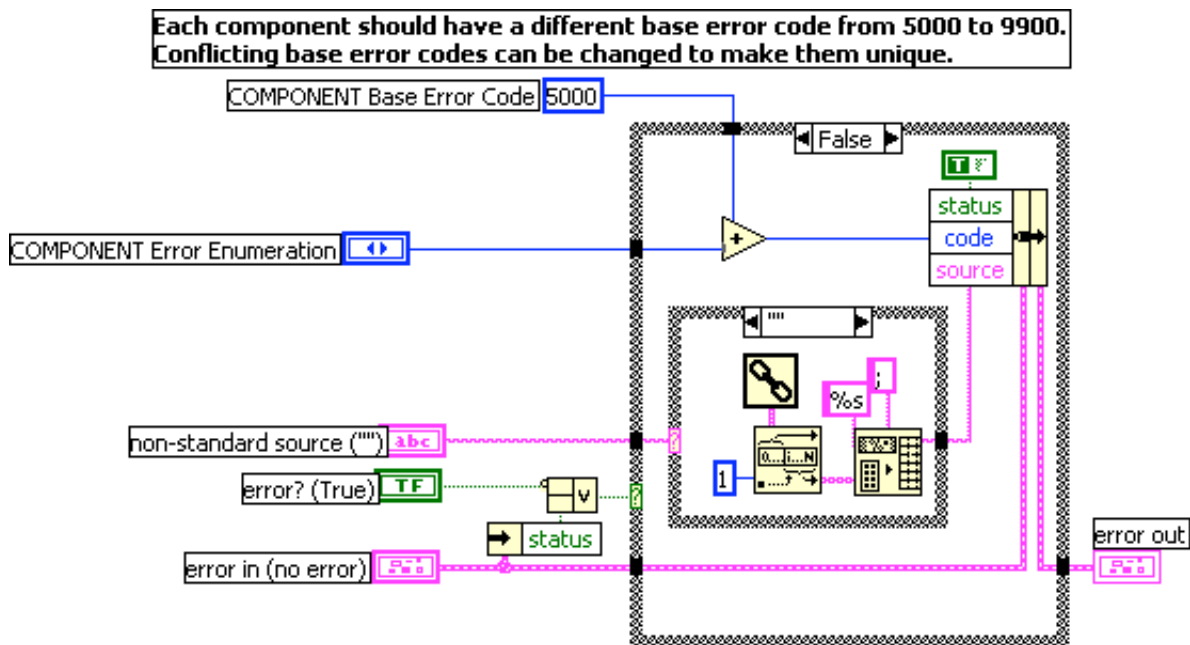


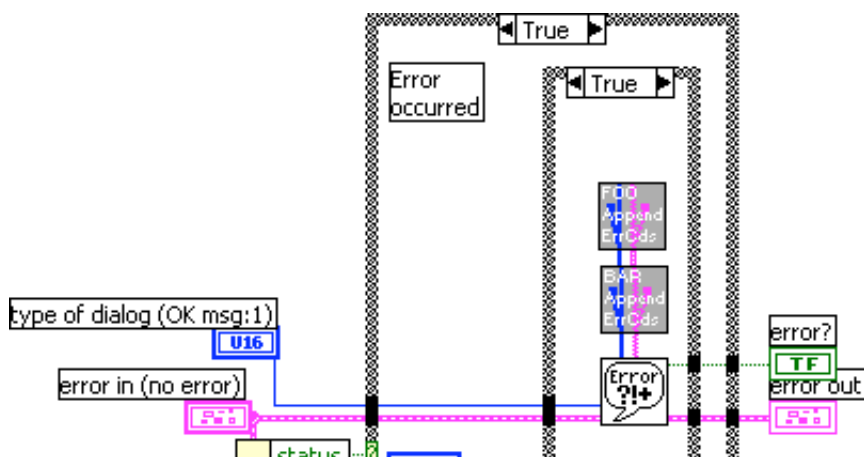
Figure 6. This is the block diagram for the LabVIEW 5 version of *COMPONENT Insert Error.vi*. By default, the source string is generated from the calling chain. A base error code is added to the error enumerator to offset the code into a unique section of the user-defined range, 5000-9999.

Building Error Clusters with Windows Codes

For users of LabVIEW under Windows, an additional VI is provided for dealing with Windows' built-in error codes. These codes are only encountered when making direct calls into the Windows dlls. *Insert Win32 API Error.vi* calls the Windows *GetLastError* function to retrieve an error code, so it doesn't have an explicit error code input.

Error Handler for User-Defined Codes

Once you've started building error clusters with user-defined codes, you'll need to employ *General Error Handler.vi* rather than *Simple Error Handler.vi*. The error handler template provided here, *PROJECT Error Handler.vit*, builds on *General Error Handler.vi* by automatically constructing input arrays of codes and messages when needed. As an example, *FOOBAR Error Handler.vi* (Figure 7) shows how the template can be used with two components, FOO and BAR. *FOO Append Error Codes.vi* generates the codes and messages for the FOO component, *BAR Append Error Codes.vi* concatenates the BAR component arrays, and the combined arrays are passed to *General Error Handler.vi*.



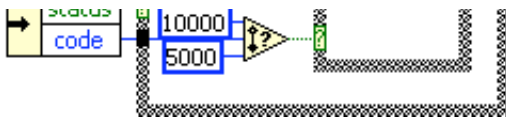


Figure 7. This example shows an error handler for a project with two components. The case structures shorten the execution time of the VI but don't otherwise affect its behavior.

You may note that the behavior of *FOOBAR Error Handler.vi* would be unaffected by removal of its two case structures, because *General Error Handler.vi* itself contains the same logic. However, the performance of the VI as written is substantially better than it would be without the case structures. Since both the construction of the error tables and *General Error Handler.vi* can be significant consumers of CPU time, this error handler skips them whenever possible.

For users of *Insert Win32 API Error.vi*, there's also *PROJECT+WIN32 Error Handler.vit* which adds handling of Win32 error codes to the functionality of *PROJECT Error Handler.vit*.

Instantiating the Templates

The tricky part of using this error handling system is the initial setup of the files. The word "COMPONENT" in multiple file names and control names must be replaced without breaking linkages. To do this in LabVIEW 5.1, a set of step-by-step instructions are provided in an HTML file on the resource disk. The instructions use the Moore Good Ideas renaming tool (provided) which automatically renames a cross-linked group of files. The control renaming must then be performed manually. With LabVIEW 6.0 controls can be renamed programmatically, so *Instantiate COMPONENT Template.vi* performs all the tasks found in the step-by-step instructions. (Be sure to copy the Error Handling directory to a writable disk before using *Instantiate COMPONENT Template.vi*, since a subdirectory is created to hold the output files).

PROJECT Error Handler.vit and *PROJECT+WIN32 Error Handler.vit* are LabVIEW template VIs, so you just open them to make copies. Each contains a single call to *COMPONENT Append Error Codes.vi* which must be replaced.

Conclusion

Error handling is an essential part of creating a robust program. The error cluster mechanism in LabVIEW can be used to create relatively simple block diagrams, but some work is required to provide user-defined error codes. The scheme presented here requires some initial setup time, but then makes using and modifying error codes a breeze. Performance improvement and the ability to remove code number conflicts easily are side benefits of the scheme.

[Back to Error Handling Downloads](#)