

User guide for the Error & Warning LabVIEW toolset

Rev. 2014

December 2nd 2014

1	INTRODUCTION	1
2	THE LABVIEW ERROR CLUSTER.....	2
2.1	The error description.....	3
2.2	Custom error descriptions.....	4
3	ERROR & WARNING TOOLSET	5
3.1	Setting errors.....	5
3.2	Defining errors.....	7
3.3	Appending errors.....	8
4	DEPENDENCIES	9
5	LICENSING.....	9

1 Introduction

The Error & Warning LabVIEW toolset is a collection of VIs for setting, converting and modifying errors and warnings:

- Set and clear errors and warnings.
- Filter error codes.
- Dynamically define custom global error descriptions at runtime.
- Extract various error parameters including error description and explanation.
- Append errors and warnings (as opposed to the built-in merge errors which potentially lose error information).
- ...all in VIs with really compact icons for easy inclusion in your block diagrams.

This toolset has been developed with LabVIEW 2012 SP1 but should be compatible with both newer and older LabVIEW versions as well. See section 5 “Licensing” for terms of use.

2 The LabVIEW error cluster

The LabVIEW error cluster is probably one of the most familiar objects in LabVIEW. However, there might still be some useful information in this chapter, so please read on as the following explains how this toolset interprets the data on the error wire – especially the **source** field.



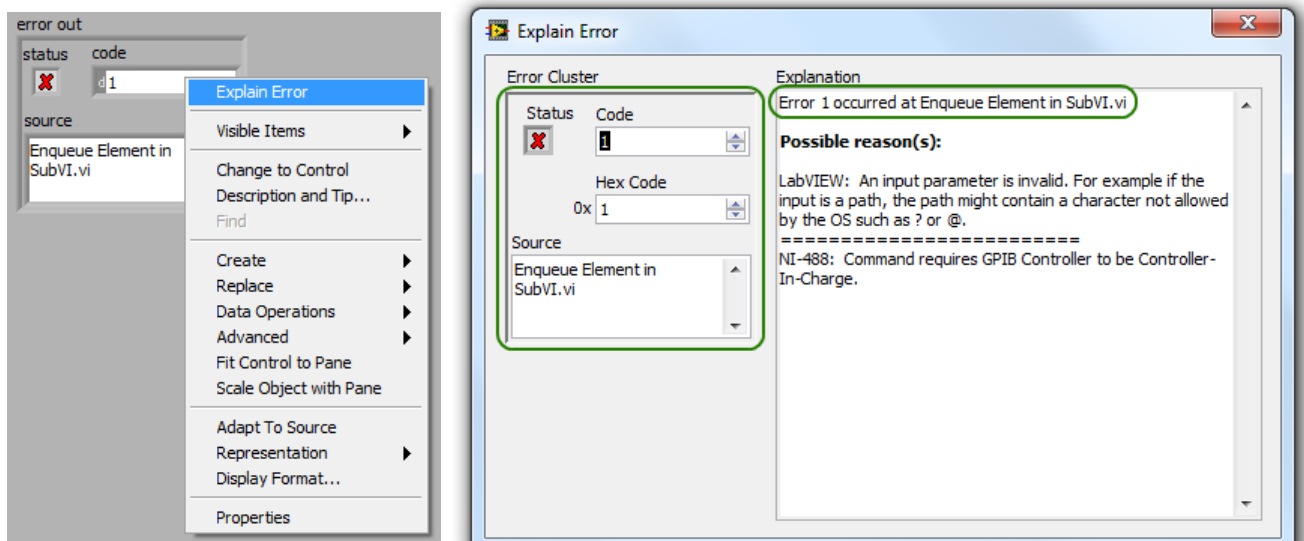
The error cluster contains three fields; **status** (a Boolean), **code** (an I32) and **source** (a string). The two first, **status** and **code**, are straightforward: **code** holds the current error or warning code. When **status** is True the cluster contains an error, and when **status** is False the cluster contains a warning *if and only if* **code** is also non-zero. A common misconception among LabVIEW programmers is that warnings have negative codes and errors have positive codes, but that isn't the definition used in LabVIEW. When **code** is zero and **status** is False this is equivalent to "No error". The ErrorType function of this toolset returns the error type from the error cluster:



There are some **code** ranges reserved for user defined errors. These ranges might change between LabVIEW versions, but as of this writing they are:

-8,999 through -8,000
5,000 through 9,999
500,000 through 599,999

The **source** field usually contains exactly that – the source of the error. For many (but not all) of the built-in LabVIEW functions this will be the call chain including the name of the function block that produced the error. The example below is the error output from an Enqueue Element primitive with an invalid queue reference wired to it (right-click on an error control and select Explain Error to get an error explanation):

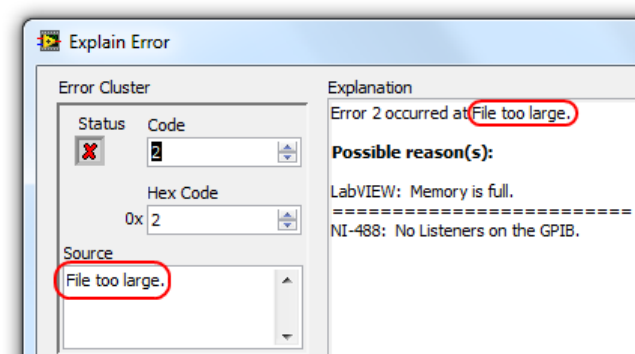


Notice how the explanation is built from the elements of the error cluster: Essentially it is “Error” (or “Warning”, depending on **status**), then the **code**, then “occurred at” and then the **source** string. If you need this string in your program you can use the GetExplanation function of this toolset to get it:

Error in 

2.1 The error description

Everything after “Possible reason(s):” in the explanation is called the error **description**¹, the specific value of which depends on **code**. For built-in errors the **description** isn’t contained within the error cluster but comes from a database in LabVIEW. Sometimes LabVIEW programmers use the **source** field of the error cluster to contain an error description when throwing custom errors, but leave out the source. This obviously yields meaningless explanations:

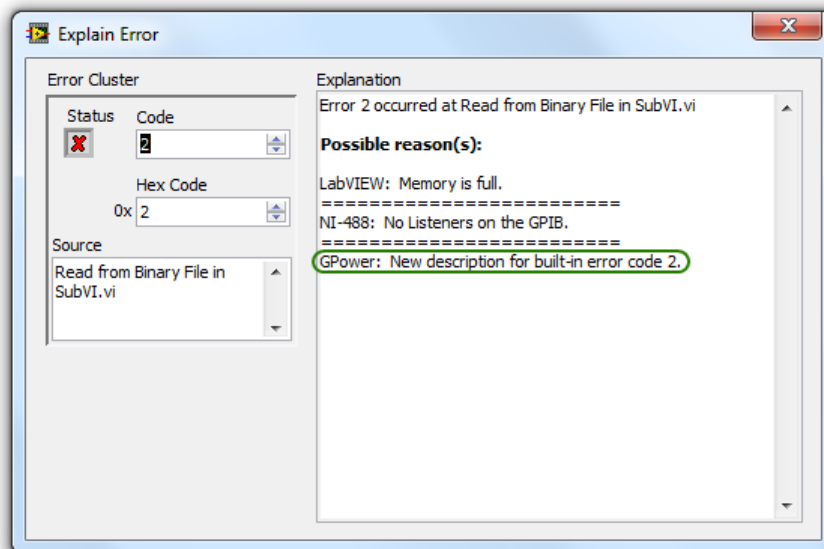
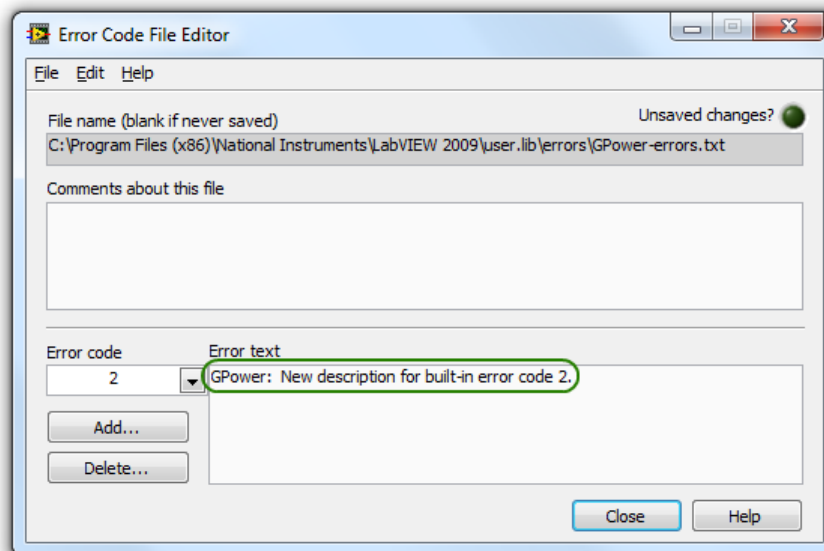


It would be optimum if **source** was always the call chain, and thus the proper place for the description would be in the possible reason(s) section. But there is no built-in way to add error descriptions to the database at runtime nor is there a field in the error cluster for descriptions, right? Or is there? More on that in the next section...

¹ Some of the built-in error handling functions calls this part a “reason”, a “message”, or even simply “error text”. I stick with “**description**” within this document and throughout the toolset.

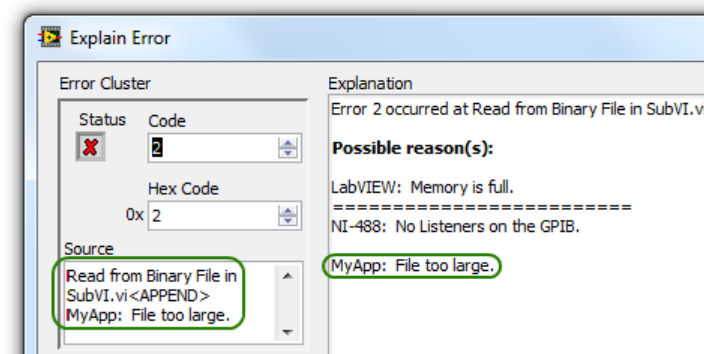
2.2 Custom error descriptions

It is possible to define custom descriptions, both for built-in and user defined error codes, with the editor found at Tools→Advanced→Edit Error Codes. This editor creates and modifies XML-files that LabVIEW reads when it launches. If you define a custom description for a built-in error code, or create multiple descriptions for a user defined error code, all descriptions will be appended and presented together when you select Explain Error:

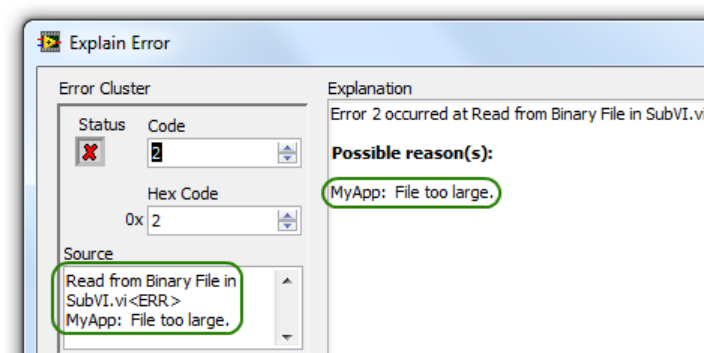


You can bundle such XML-files with your own application, but since these files are only read when LabVIEW launches they are static definitions you can't modify at runtime. The **source** string supports two tags that enable you to set a custom description at runtime though: the <APPEND> and the <ERR > tags.

Everything following an <APPEND> tag in the **source** string will be appended to the existing descriptions when you build an explanation string, anything preceding the <APPEND> tag will be classified as the source part:

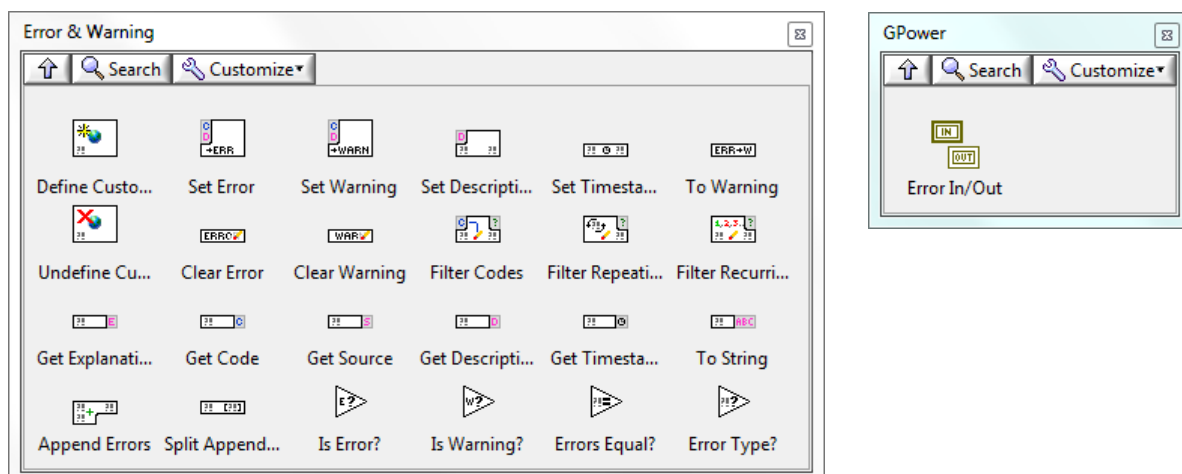


Everything following an <ERR> tag in the **source** string will replace any existing descriptions when you build an explanation string, anything preceding the <ERR> tag will be classified as the source part:



It seems cumbersome to format source strings the proper way to get an error described just right, doesn't it? This toolset makes that a breeze – please read on.

3 Error & Warning toolset



The Error & Warning toolset contains 24 block diagram functions and a single front panel control. The Error In/Out “control” drops both an ‘Error in’ control and an ‘Error out’ indicator on the front panel when dragged and dropped there. The most important functions are explained in detail in the next sections.

3.1 Setting errors

It's quite ungainly to format your source strings with call chains as well as <APPEND> and <ERR > tags every time you want to throw a custom error or you want to add some specific information to a built-in

Error code

Error description ("")

Error in (no error)

Error label ("Custom")

Append timestamp? (F)

ERR

Error out

Warning code

Warning description ("")

Error in (no error)

Warning label ("Custom")

Append timestamp? (F)

WARN

Error out

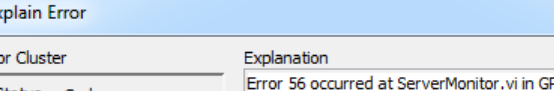
Description

Error in

Label ("Custom")


25 28

Error out



Explain Error

Error Cluster

Status	Code
	56

Hex Code: 0x38

Source

ServerMonitor.vi in GPApp.vi
<APPEND>=====

Explanation

Error 56 occurred at ServerMonitor.vi in GPApp.vi

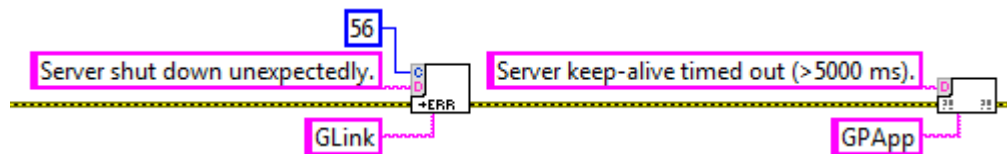
Possible reason(s):

LabVIEW: The network operation exceeded the user-specified or system time limit.
=====

GLink: Server shut down unexpectedly.
=====

GPApp: Server keep-alive timed out (>5000 ms).

6 of 10



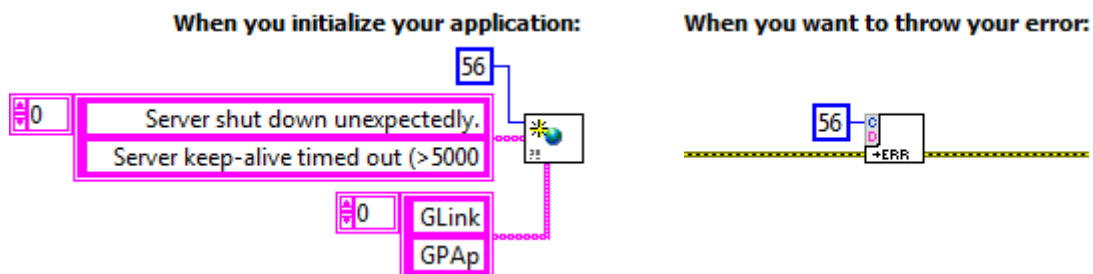
Even though the above makes it quite easy to set correctly formatted errors, it can get simpler yet – the next section is about defining global error descriptions.

3.2 Defining errors

Setting custom errors one description at a time can still be unsuitable if you want to throw the same custom error from several locations in your code, as you then face the challenge of maintaining identical description strings for identical error codes for instance. LabVIEW lets you bundle custom error code files with your application, but since there is no way to make LabVIEW *reload* these error code files after your application has launched, this toolset offers its own dynamic error description registration methods at runtime. They are DefineCustomCode and UndefineCustomCode:

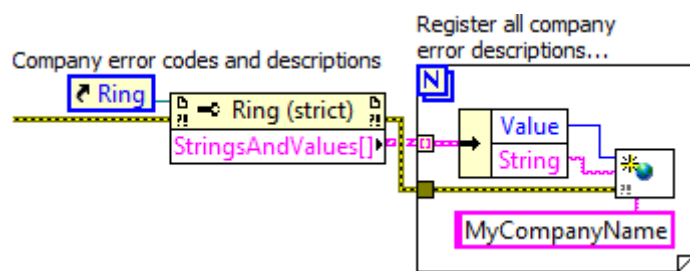


When you define custom descriptions using this method they instantly become available in the entire LabVIEW instance just as if they were defined statically in an error code file – the only requirement is that you use this toolset to set your errors. These two code snippets will give you the same error string as illustrated in Figure 1 in the previous section:



You can of course still add local descriptions to your errors in addition to what's defined globally.

Using DefineCustumCode makes it simple to define all your personal or all your company's custom error descriptions with a single reuse VI. For example like this (such a code snippet could also read the codes and descriptions from a file or from a database of course):

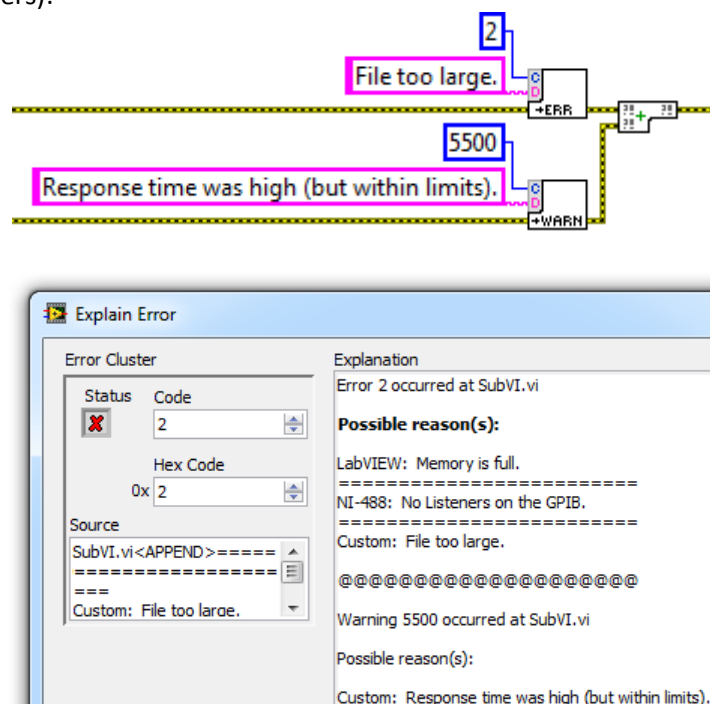


You don't have to worry about calling the above code more than once per LabVIEW instance (if it was called in all of your applications that might run simultaneously in the same LabVIEW instance for example) as the DefineCustomCode function doesn't register duplicate code/description pairs.

3.3 Appending errors

LabVIEW offers the familiar built-in function for merging errors which basically takes in a number of error clusters, returns the first encountered error or warning among these and discards the rest. If you want to keep more than one error cluster from within a subVI you'll have to return an array of errors from it. This isn't hard to do, but it breaks the simplicity of the error wire when you suddenly need to wrench an array of errors around your code which none of your usual subVIs are able to take as input on their **error in**. Even harder than designing in an array of error clusters from the start is when you want to add it to a large existing application. That will certainly lead to changes to all downstream code.

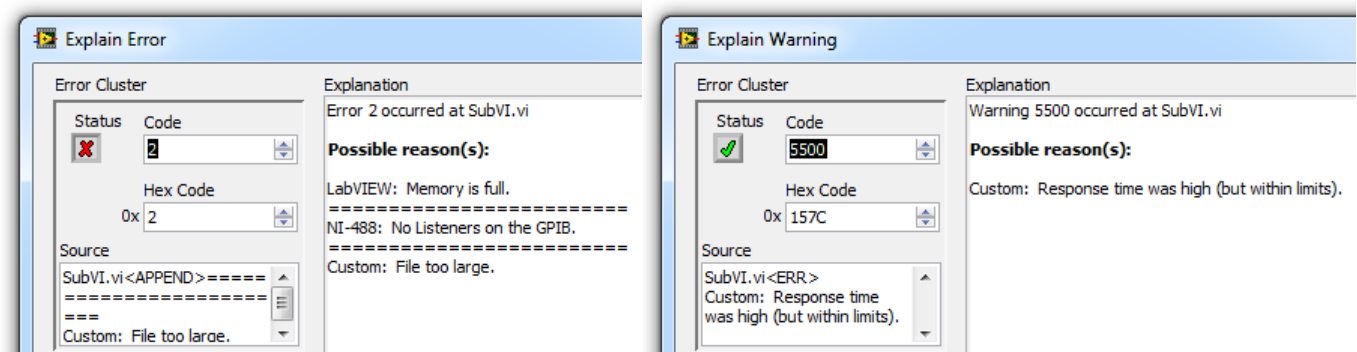
Therefore this toolset offers an AppendErrors function which lets you maintain any number of errors and warnings within *a single* error cluster without losing any information. This example below shows an error and a warning appended (the AppendErrors function is polymorphic so it also accepts appending an array of error clusters):



The AppendErrors function appends only errors and warnings; it omits 'No error' error clusters. By default it appends all errors and warnings unconditionally but you can configure it to only append unique errors thus filtering out duplicate errors.

If you want to operate on appended errors and warnings individually you can use the SplitAppended function to convert the appended error cluster into an array of error clusters each containing only a single error or warning as usual:

Appended errors in  Errors out



4 Dependencies

This toolset has no external dependencies.

5 Licensing

This toolset is free and open source, and is covered by a BSD 2-clause license (see the Error_License.txt file included with the toolset). But why a license when it's free? The BSD license is one of the most unrestrictive licenses out there, and we use it mainly to disclaim our responsibility for that software which it covers. So you may use it for whatever you want, but we won't take any responsibility if it blows up on you. It by all probability won't blow up on you (we use this software ourselves), but if it does we won't take responsibility for your losses. We certainly would be glad to hear about any problems you might have run into though, so we can fix those issues. The BSD license in layman's terms grants you these rights:

- You may use this GPower software for free for both commercial and non-commercial applications. This includes both development of and inclusion in such applications.
- You are at liberty to employ any licensing scheme you wish on your final application. That this GPower software is open source and free does not mean your application using it has to neither be open source nor free. Your application can employ a proprietary license, you can charge money for it, it can be closed source etc.
- You are allowed to change the source code of this GPower software as you wish with no obligation to publish your changes in any way.

The BSD license does put two obligations on your shoulders though, even if you use this GPower software in modified form:

- You **must** leave the GPower copyright notice on the front panels of the VIs that make up the software. It will look something like this:

Copyright © 2014
GPower
See enclosed license file for terms and conditions

It is not required that you include the front panels of the VIs of this GPower software in your built application. But if you do, or if you bundle the source code, then you must leave the mentioned copyright notice in place. If you're not including the front panels of the GPower software, then the copyright notices from those front panels obviously can't be present.

- You **must** preserve the unmodified license text file with the software, and it must reside somewhere your end-user would expect to find it (this is to preserve the disclaimer). If you include the GPower software source code you can just leave the license text file with those VIs. If you deliver binaries exclusively you can for instance save the license text file together with your other material rights documents, or you could paste the license text into the small-print section of your user guide.