

Style Guide

DQMH Consortium

Version release/v1.0.1, 2022-11-10: 1954a5c1



Table of Contents

1. Introduction	1
2. Scope.....	2
3. References	3
3.1. Reference Documents	3
3.2. Abbreviations, Acronyms, And Definitions	3
4. LabVIEW environment configuration	5
5. General Software Development Style.....	6
5.1. Source Code Control	6
5.1.1. Committing files into Source Code Control.....	6
5.1.2. Pulling files out of Source Code Control.....	6
5.2. Code Re-use and VI Package Manager	6
5.3. Looping	7
5.4. Configuration Files	7
6. LabVIEW Style Guide.....	8
6.1. File organization	8
6.2. File naming	8
6.3. VI Documentation	9
6.4. Control and indicator documentation.....	10
6.5. Block diagram documentation	11
6.6. Class Documentation.....	12
6.7. Library Documentation	13
6.8. VI Icons and connector panes	13
6.9. Graphical User Interface (GUI) Design.....	15
6.10. Front Panel Design	16
6.11. Dialog-Style VIs	18
6.12. Error handling	18
6.13. Error logging	18
6.14. Error codes.....	19
6.15. Block diagrams	19
6.16. General programming style	21
6.16.1. Architecture.....	21
6.16.2. LabVIEW environment settings	21
6.16.3. Case structures.....	21
6.16.4. Appropriate use of data types.....	22
6.16.5. Data coercion	22
6.16.6. Event Handling.....	23
6.16.7. Performance Considerations	23
7. VI Analyzer tool.....	25

8. DQMH.....	26
9. Legal Information	27

Chapter 1. Introduction

This document identifies robust style and programming best practices that should be used when developing software in LabVIEW programming environments.

Developers should also reference this document when conducting code reviews, and new hires should consider this an overview of the required coding standards.



This document is licensed under the BSD 3-Clause License. Feel free to use this document as a springboard for your own style guides. You are allowed to copy the contents and use them in any way you see fit.



The document sources are available in this [GitLab Project](#)

License

BSD 3-Clause License


Copyright (C) 2022, DQMH Consortium
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:


1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Chapter 2. Scope

Items highlighted with  ship with the NI VI Analyzer.

Items highlighted with  are available in the LabVIEW Example Style VI Analyzer Test Suite, available [here](#).

Items highlighted with  are included as tests in the [DQMH Consortium VI Analyzer Tests Suite](#) or have scripting tools available to enforce their style.

You MUST review items highlighted with  before adapting this document for customer use.

Chapter 3. References

3.1. Reference Documents

Ref	Document	Doc Number	Issue
RD1	NI LabVIEW Help> LabVIEW Style Checklist	https://www.ni.com/docs/en-US/bundle/labview/page/lvdevconcepts/checklist.html	Latest
RD2	Example Style Guidelines	https://forums.ni.com/t5/Developer-Center-Resources/Example-Style-Guidelines-LV-Add-ons/ta-p/3510533	Latest

3.2. Abbreviations, Acronyms, And Definitions

Abbreviation	Description
ADE	Application Development Environment
API	Application Programming Interface
BD	Block Diagram (LabVIEW)
DLL	Dynamic Link Library
FP	Front Panel (LabVIEW)
GUI	Graphical User Interface
LVLIB	LabVIEW Project Library File
LVPROJ	LabVIEW Project File
LVOOP	LabVIEW Object Oriented Programming
PPL	Packed Project Library
VIPM	VI Package Manager
VIPC	VI Package Configuration
LV	LabVIEW
VI	Virtual Instrument

Term	Definition
PascalCase	No spaces between words. First letter of each word is uppercase. Acronyms are all uppercase. For example, SequenceFile.
Spaced Pascal Case	Spaces between words. First letter of each word is uppercase. Acronyms are all uppercase. For example, Sequence File.
camelCase	No spaces between words. First word is all lower case. For all other words, first letter is uppercase. For example, sequenceFile.

Term	Definition
lower case	All lowercase with spaces between words. For example, sequence file.
Snake_Case_Caps	Underscore between words. First letter of each word is uppercase. Acronyms are all uppercase. For example, Sequence_File

Chapter 4. LabVIEW environment configuration

★ In order to promote consistent development practices and style, we strongly recommend adding the following LabVIEW ini tokens to the LabVIEW ini file.

```
QuickBold=True ①  
  
bookmarkmanager.showvilib=True  
  
bookmarkmanager.showresource=True  
  
defaultErrorHandlingForNewVIs=False  
  
sourceOnlyDefaultForNewVIs=True  
  
QuickDropFastSearch=True  
  
reqdTermsByDefault=True  
  
defaultControlStyle=0  
  
FancyFPTerms=False  
  
defaultControlStyle =0
```

① Since LabVIEW 2018, this key is no longer necessary

Chapter 5. General Software Development Style

5.1. Source Code Control

Keep all source code under revision control using Git.

- Decide on a strategy for maintaining dependencies (either in your repo or in a central location) and stick with it for each project your team works on.
- Make sure that your repo contains a `README` file which also provides setup instructions and specifies where to find dependencies.
- Consider having a project-specific `.vipc` file in your repository.
- Do not keep binary dependency files like installers in your repo.
- SCC applications are designed to manage shared development of the code.
- However, using SCC is also beneficial to a single user as it allows for changes to be undone and code to be reverted to a previous version.
- Descriptive comments with each commit help identify when a bug was introduced.
- Source Code Control should be used for all projects no matter the size.

5.1.1. Committing files into Source Code Control

- When committing files into source code control always adds a descriptive comment describing why you modified the files. This makes changes and their effect easier to track.
- Commit meaningful groups of files to make each commit as small as possible and only related to one modification action (feature addition, refactoring, bug fixing...)

5.1.2. Pulling files out of Source Code Control

- Always ensure you have synchronized your local repository with the latest code version of your working branch before commencing work to ensure you are not working with old code.

5.2. Code Re-use and VI Package Manager

- When writing software, reuse existing software wherever possible. Using common libraries of code can dramatically reduce both development time and the risk of programming errors.
- Browse through your reuse library before developing a new application (preferably using a tool such as VIPM) to ensure you are not reinventing the wheel.
- Another added benefit of this is the constant review and refinement of the code in the library or repository by other developers as they reuse it. This leads to improved quality.
- Ensure that any code placed into a library or reuse repository is fit for use and fully validated. Otherwise, plan the necessary validation. Un-validated code will harm the value of your reuse repository and limit its usefulness.

5.3. Looping

- Whenever possible, avoid duplicating code “end on end.” Use loops to perform tasks multiple times in sequence.
- Consider using array data to store variables when working with loops as LabVIEW allows easy array indexing.




Since LabVIEW 2019 Set and Map are also a good way to store variables.

- Make sure you consider the side effects of using loops as they could affect time-critical applications.
- When using While Loops ensure that it is always possible to stop the loop in an appropriate, safe manner and that no resources are left in an unsafe state.
- Shift registers should be used for wiring values through a `for loop` to prevent default values to be returned for any output when the `for loop` iterates zero times.

5.4. Configuration Files

- Avoid using constants that are likely to change. Use configuration files to store all variables and configurations used by the application.
- Use configuration files to alter the way an application behaves.

Chapter 6. LabVIEW Style Guide

- Write neat diagrams. They are easier to maintain and debug.
-  Avoid extraneous use of Local and Global variables. Wherever possible, pass values by wire.
- Add documentation. Something that you understand well today might not be as clear in the future or to another developer.
- Make sure VIs are tested on each OS they are to be used on. This is especially important when writing code that interacts with the local file system where there may be significant differences between platforms.

6.1. File organization


- Organize project source code libraries (.lvlibs) on disk similarly to the project organization.
- All members of a library (.lvlib) should reside in the same path on the disk as the .lvlib file itself. The only organization on disk that should be done is that nested libraries should be in their own folders on the disk.



A flat file organization on disk for .lvlib member VIs is easier to maintain. The organization should only take place in the project.

- Never using LLBs to store your source code. LLBs are a legacy technology and do not provide many of the benefits and protections that project libraries (.lvlib) offer. There are specific use cases where an LLB is a legitimate choice, e.g., when minimizing load time of a collection of VIs is important or for source distribution
- If your project consists of many VIs grouped logically then consider creating a top-level project (.lvproj) containing several libraries (.lvlib) as this will help developers navigate the project more easily.
- All subVIs listed in a project (.lvproj) should reside in virtual folders.
- Avoid Auto-Populating folders.

6.2. File naming

- Use English language if possible. English is very concise, and you can get help more easily on public forums, and make it easier to share code.
- Unless you have a good reason not to do it, name **ALL** files (VIs, Libraries, Projects, etc.) using Spaced Pascal Case. (e.g., `Application Management.lvlib`)
- Give VIs meaningful names without special characters, such as backslash `\`, slash `/`, colon `:` and tilde `~`.
- Consider organizing your subVIs in libraries. The name spacing provided by libraries is preferable to applying prefixes to the file names of related VIs.
-  When creating typedefs use the following convention.

```
X--enum.ct1  
X--cluster.ct1
```

```
X Argument--clusterctl
X--mapctl
X--setctl
```

★ The main benefit of this naming convention relates to Quick Drop. Since the typedefs are in the project, you can type (as an example) “--enum” in Quick Drop to get the names of all enum typedefs in the project, which can help if you do not know the exact name of the enum typedef you need.

- When using `variable constants` (i.e., constant values wrapped inside subVIs, or “constant VIs”), name them `X--constant.vi`.
- When using paths that are used in different sections of the project wrap them in a constant VI and name them `X path--constant.vi`

Constant VIs are used because their values cannot be changed at runtime. This makes them preferable to global VIs. Even if the author of the code intends for his global value to be written zero or one time, this intention is not enforceable.

6.3. VI Documentation

- Write a VI description (**File › VI Properties › Description**), proofread it, and check its appearance in the **Context Help** (`Ctrl` + `H`) window.



This description supports the `` HTML tag to highlight important text).

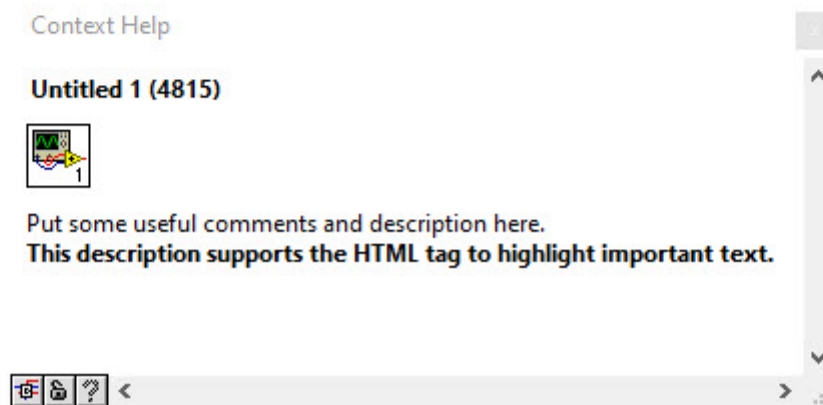
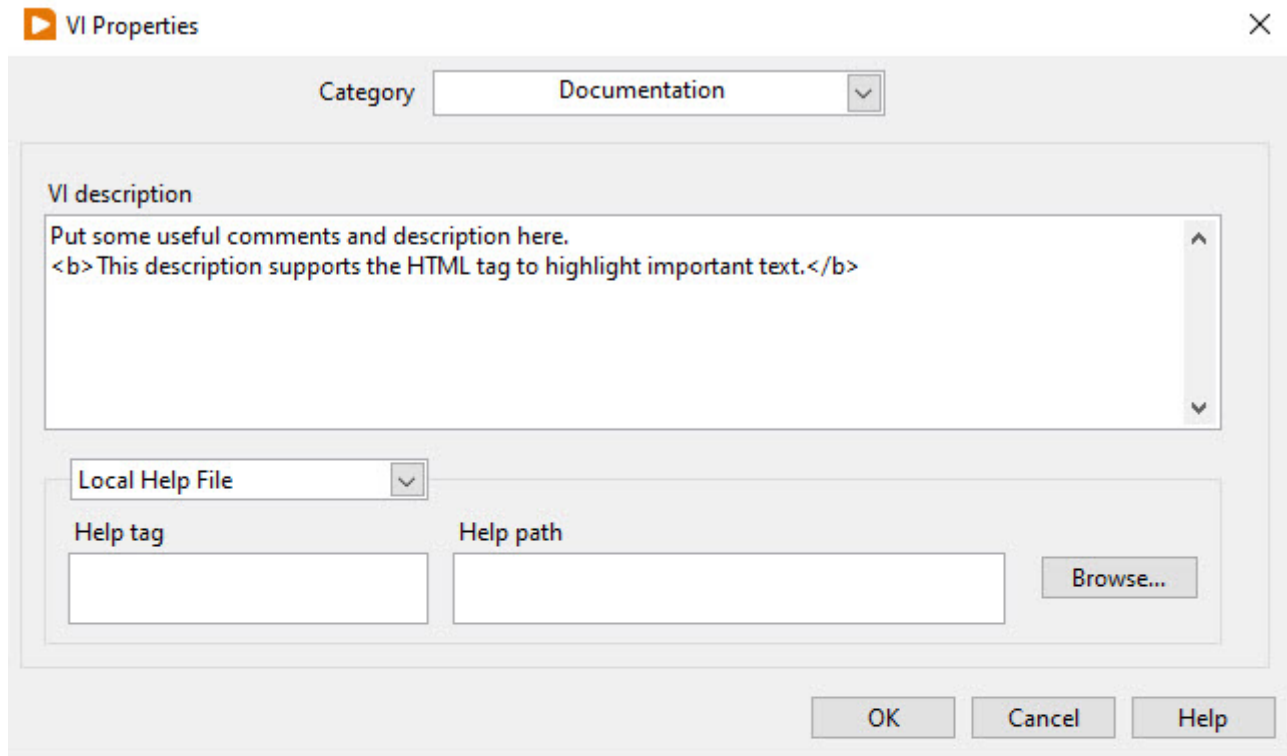


Figure 1. VI properties window

6.4. Control and indicator documentation

When creating public APIs, document front panel controls and indicators by right-clicking on them and selecting 'Description and Tip' from the popup menu.

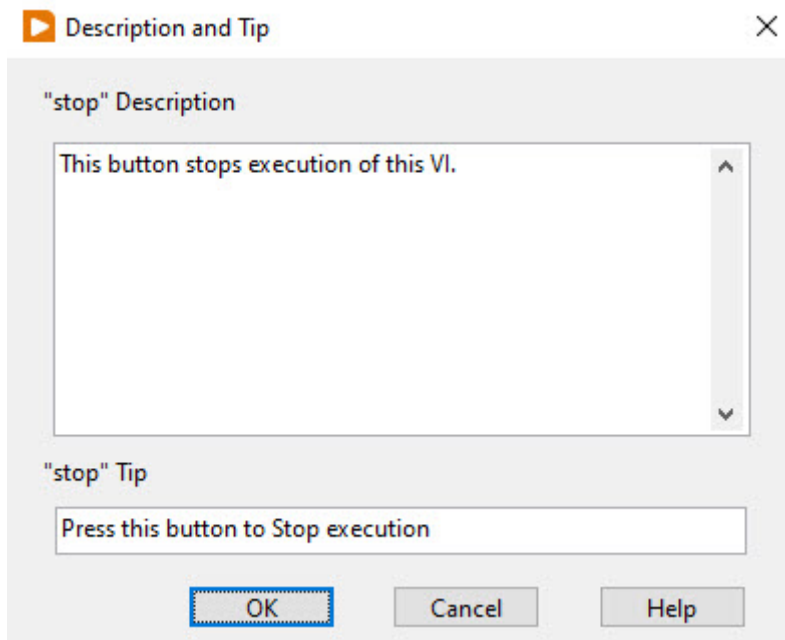


Figure 2. Control description and tip window

6.5. Block diagram documentation

- On the block diagram of a VI, add text notes like the ones shown below, and use the default free label color (light yellow) for all comments.

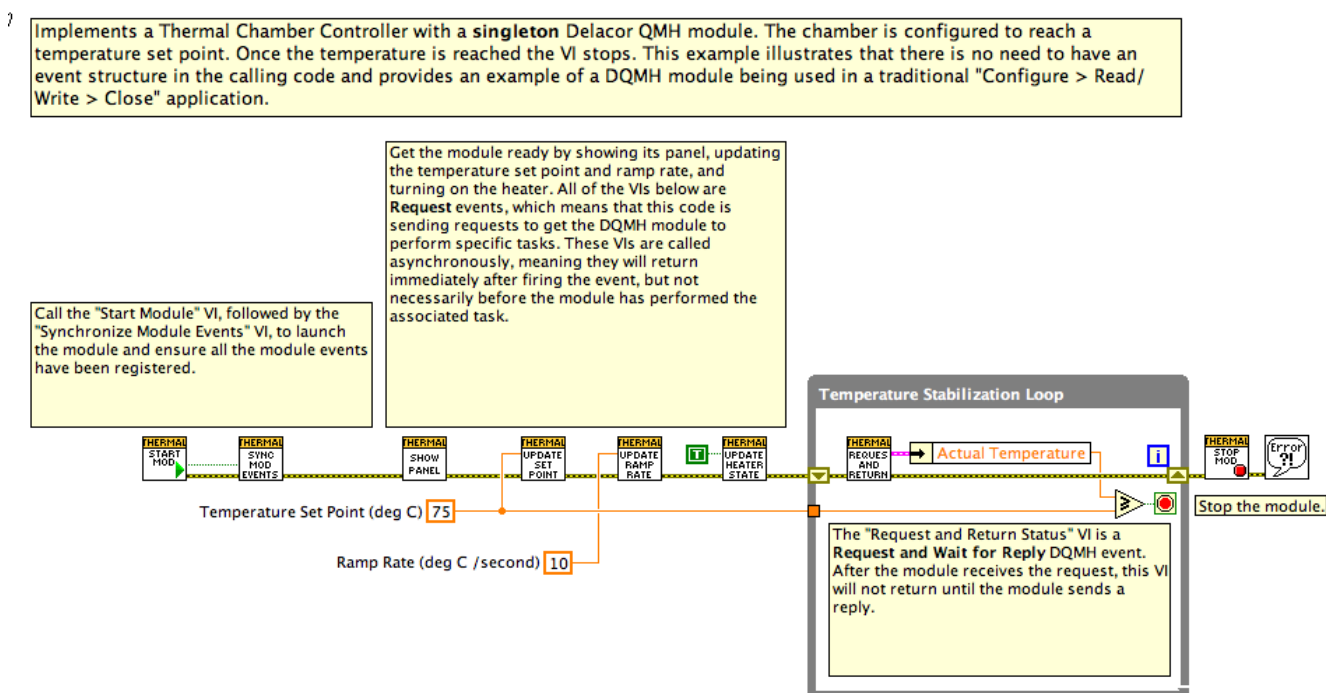


Figure 3. VI diagram comments

- Label long wires to aid code readability. Show the wire label by right-clicking on the wire and selecting **Visible Items > Label**.

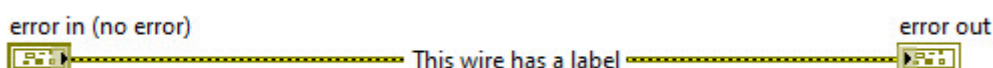


Figure 4. Wire label

- Connect comments to code when possible. It improves comment understanding.

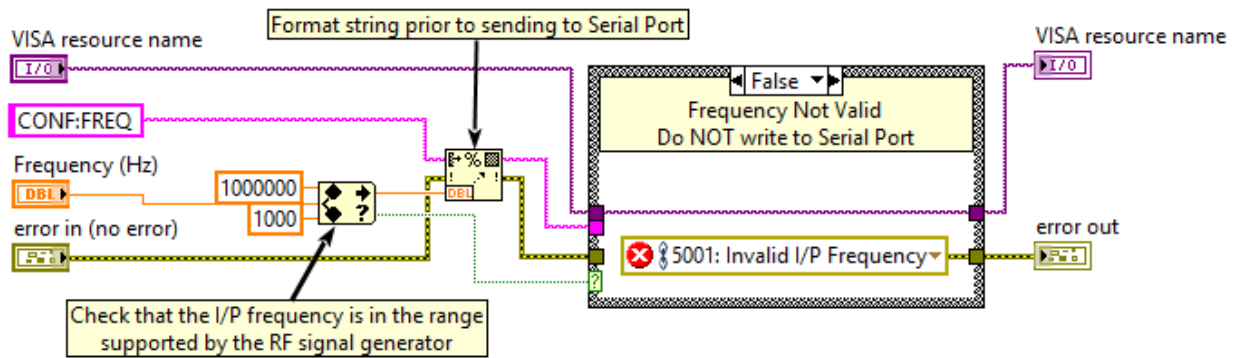


Figure 5. Attached comments

- Use scrollable string comments for especially long comments such as credits, patents or legal notices.

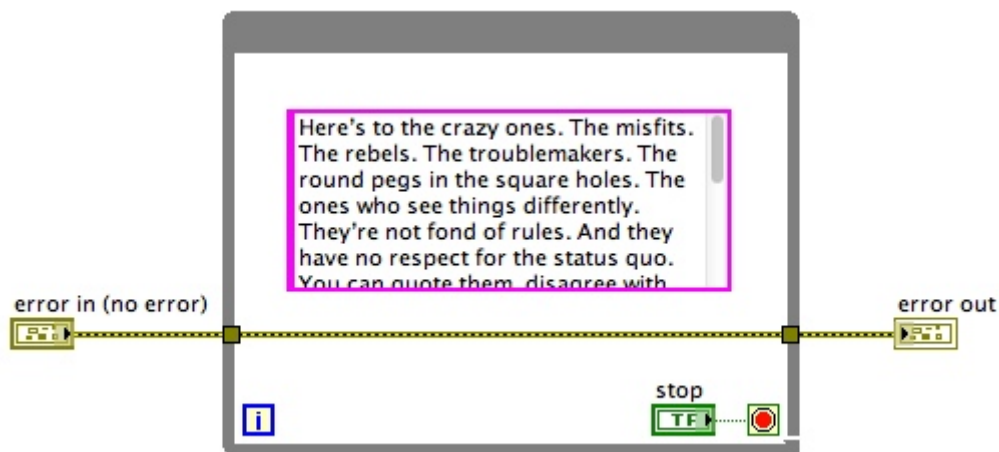


Figure 6. Scrollable string comments

- Subdiagram labels provide a convenient method of documenting individual frames of structures.

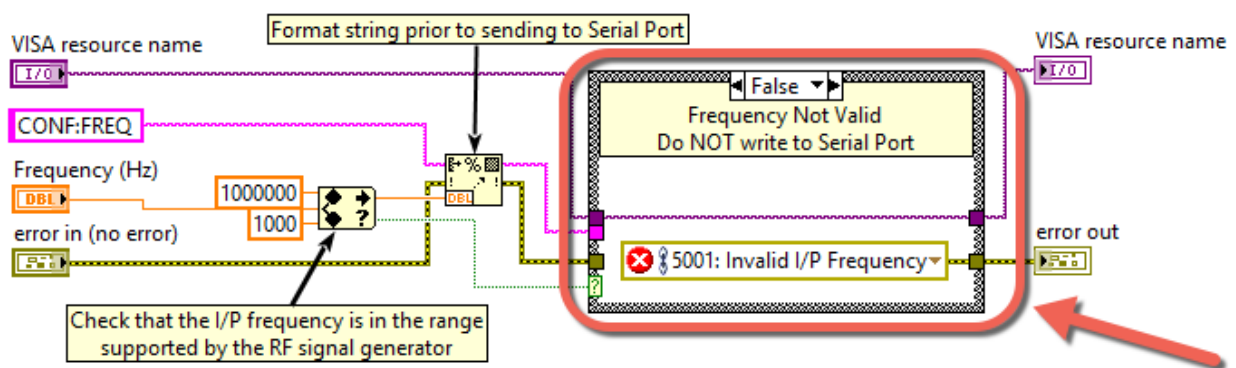


Figure 7. Subdiagram label

6.6. Class Documentation

- Make sure to add documentation to the class in its properties.



Class documentation should, at least, describe its single role purpose.



If you are using [Antidoc](#), class description content will be used to populate the documentation generated.

- Set Class names using the class **Properties >> Documentation** option and remove the `.lvclass` extension from the **Localized Name**. This makes class property nodes significantly smaller saving block diagram space.

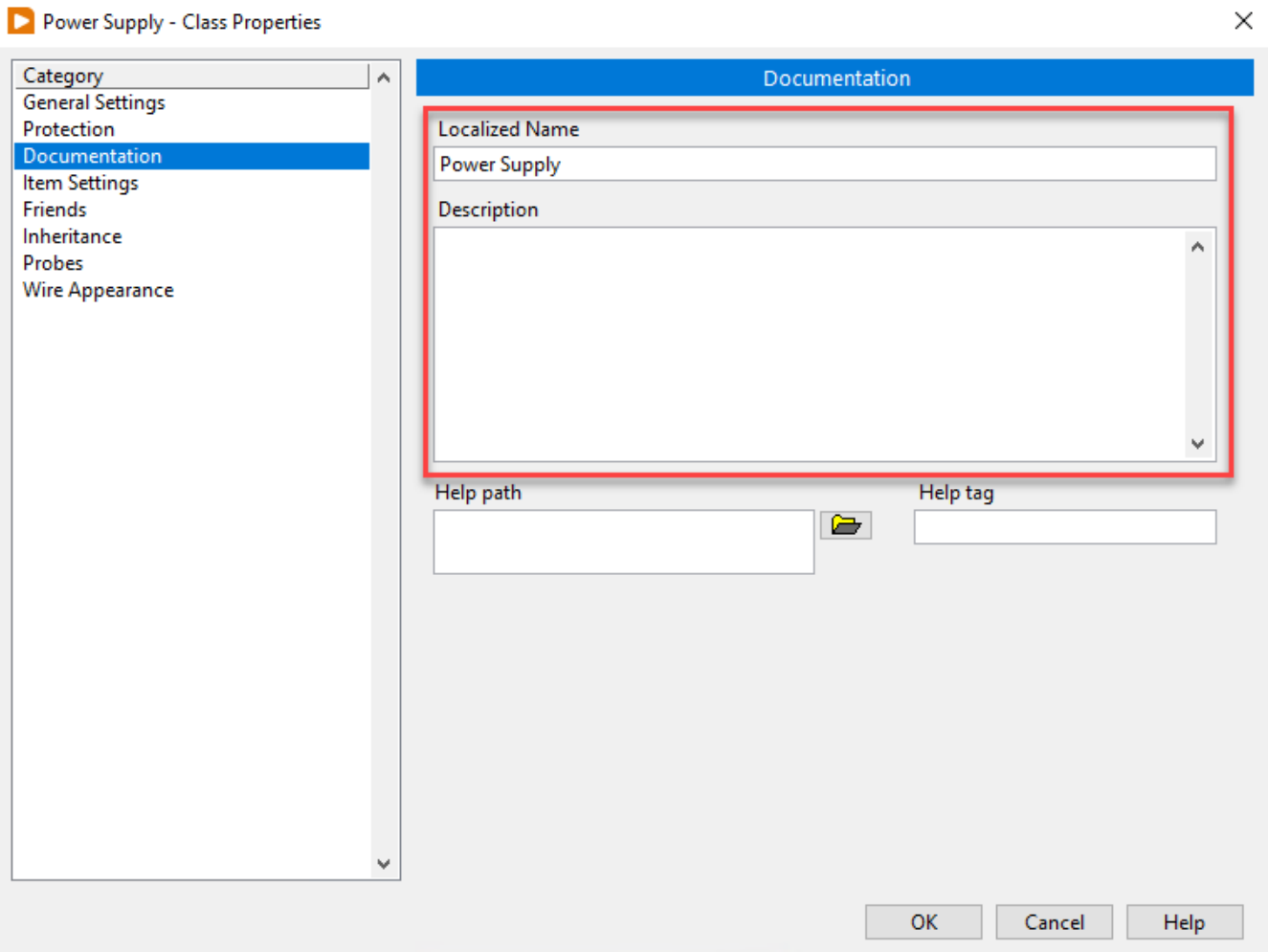


Figure 8. Class documentation window

6.7. Library Documentation

- Make sure to add documentation to the library in its properties.



If you are using [Antidoc](#), library description content will be used to populate the documentation generated.

6.8. VI Icons and connector panes

- For public API available in palettes, consider investing time in designing graphical icons.




Note the visual similarity of logical groups of functions

Figure 9. Set of VI icons

- For all other VIs, text-based icons with the library banner suffice.



In LabVIEW 2020 and later you can use the Set Text Icon shortcut from Quick Drop (Press **Ctrl** + **Space**, then **Ctrl** + **K**) to assign a text-based icon to the VI based on its file name. You can also type space-delimited text in Quick Drop before pressing Ctrl-K to use that text in the icon instead of the file name.

-  Choose the 4-2-2-4 connector pane pattern (default for new VIs).
- Leave empty terminals to allow for future development. Use a consistent layout across groups of related VIs.
- Avoid using connector panes with more than 12 terminals. Instead, use clusters to group controls in a logical manner.
- Arrange front panel controls in a similar fashion to their connector pane positions as shown here.



Ctrl + **U** on the front panel in LabVIEW 2019 and later will automatically arrange the front panel controls and indicators to match their positions on the connector pane:



There are also various Quick Drop shortcuts available for this, e.g., [the Natttify VI](#)

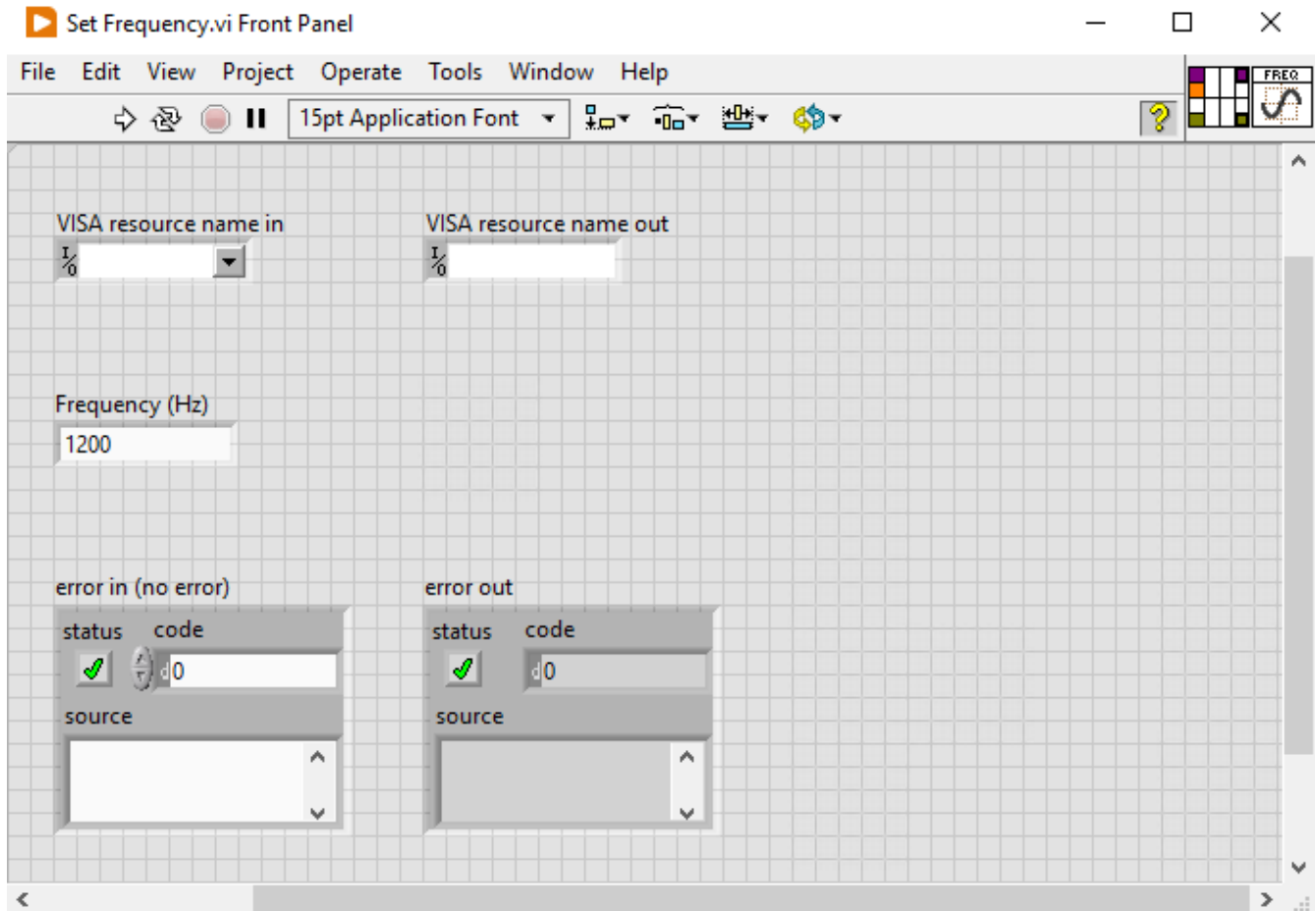


Figure 10. Front panel controls and indicators that match connector pane position

- Use Small Fonts, size 8/9, All Caps, for any icon text. This assumes you are developing content on Windows, as the Small Fonts font is different on Mac, and non-existent on Linux.
- Always use full-size (32x32) icons to ensure subVIs are easily identifiable.
- Select **recommended** or **optional** settings on the connector panes carefully. If so, make sure to set a consistent default value.
- Ensure the banners of all libraries have been applied to all VIs that are members of those libraries.

6.9. Graphical User Interface (GUI) Design

When developing GUI, consider the layout, functionality, visualization, and consistency of the GUI.

- When developing a GUI try to make the visual appearance follow the same style as any other GUIs in the same project.
- When developing for touch screen, consider sizing controls accordingly so users can operate your software with their fingers.
- Reduce the number of controls in GUIs design for production environment.
- Use tab controls to break down complicated GUIs into logical groupings. Consider using subpanels.
- Use decorations to partition the panel into logical groupings.
- Make the GUI as easy to use as possible by programmatically disabling / graying out controls and indicators that could be used incorrectly or at an inappropriate time.

Example,

If a GUI you have developed has 'Start' and 'Stop' buttons when the program is in an idle state. Disable the 'Start' button once the task has started.

Likewise, the 'Stop' button should be initially disabled until the task is running.



- If the program is waiting for critical user input, make the input GUI window modal. This ensures that the window does not get accidentally hidden behind other windows, giving the mistaken impression that the software has hung.
- Use color logically and sparingly, if at all. It is preferable to use the system colors wherever possible as they adapt to the selected desktop theme (on Windows).
- For a standalone application, ensure the front panel includes the version number. If the application is a built executable, then it is a good idea to display a version number in a suitable location such as the title bar or any corner of the top level VI.
- Place common controls such as 'Abort' and 'Continue' consistently throughout your application.
- Use decoration to border titles, functional areas and group buttons at the bottom of the GUI.
- Use default LabVIEW Application Font for controls and indicator text and labels.
- Use Vertical and Horizontal splitter bars to organize front panel items if you wish to allow resizing of the front panel, otherwise disable front panel resizing.
- Align controls wherever possible.
- Pressing the close panel button should invoke the same behavior as pressing any 'Exit' button. Use the 'Panel Close?' event.

6.10. Front Panel Design

- When replacing a control with another control of a different style, use paste-replace (drop a new control, `Ctrl` + `X` to cut, select old control, `Ctrl` + `V` to paste-replace) instead of right-click replace.
- Use Modern Controls on internal subVIs.



This setting can be enforced in the [LabVIEW environment configuration](#)

-  Use the light gray default panel background color on all non-dialog VIs.
- Do not save the Front Panel maximized.
- When creating UI front panels consider using rings instead of enums (Remember that the items in a ring are not part of its data type. Making a typedef ring does not guarantee that changes to the ring items will propagate to all typedef instances. See [here](#) for more information). If you must use an enum, make sure it is a typedef. If you use a ring on the UI, consider using an Enum to programmatically add its contents. The advantage of the ring here is the text in the ring can be less geeky or it can be localized.
-  If you are using a cluster or an enum, make sure it is a typedef. Even if you think that you will never use it outside of the VI where you placed it. Especially if the enum is used in more than one place in the same VI.

- ★ If you are using a Radio Buttons Control or a Tab Control on the front panel of a VI, make sure it is a typedef. This helps avoid the situation where a non-typedefed radio buttons control/tab control has a corresponding non-typedefed enum constant on the diagram, which can cause problems when the contents of the radio buttons control (or the names or number of pages on the tab control) changes.
- Do not use an error cluster on the front panel of the main VI to indicate an error. Instead, call a Simple Error Handler VI at the end of the diagram's execution.
- If there is a pass-through input/output pair on your VI, give control and indicator an `in` and `out` suffix, respectively. If the data is not changed, then put the `dup` suffix for the output.
- Give controls meaningful names. Use consistent capitalization.
- For UIs, display the caption instead of the label so you can show user-friendly names and it can be localized more easily.
- Make the background of control and indicator labels transparent.
- Use path controls instead of string controls to specify the location of files or directories. Set the browse options correctly (files/directories, new/existing, etc.).
- For `recommended` or `optional` input terminals, put default control values in parentheses.
- Include 'unit of measure' information in control names if applicable. For example, **Time Limit (10 seconds)**.
- Arrange controls attractively using the Align Objects and Distribute Objects pull down menus.
- Avoid overlap controls.
- Provide a mechanism to properly stop your application. It can be clicking on a stop button or by closing the front panel using the red 'X'.
- Do not use the Abort button to stop a VI as this can leave the VI (and any equipment controlled by the VI) in an undetermined state. Instead, provide appropriate 'shutdown' code in response to pressing the stop button or closing the window.
- Hide the abort button if possible.
- If you are using a Boolean control to select between two options, consider using an enumerated typedef control instead to allow for future expansion and the addition of further options. Also, an enum is easier to understand than a Boolean. (Enabled/Disabled/ vs True = Enabled and False = Disabled)
- Use `Type Def.` controls over `Strict Type Def.` controls, unless you have a control whose appearance you want to be identical across all UIs.
- Consider using one of the auto arrangement options for clusters.
- Label custom controls and typedefs with the same name as the file in which they are defined. For example, `Alarm Codes--enum.ct1` has an enum named Alarm Codes.
- ★ Single-line strings in UI VIs should be marked as 'Limit to Single Line'.
- ★ If you are using the Unit Test Framework, avoid using the following characters in control and indicator labels, as they are incompatible with UTF's .lvttest files:

```
[ - open bracket
] - close bracket
\ - backslash
```

/ - forward slash
 \n - line feed

6.11. Dialog-Style VIs

- For dialog-style VIs use the System **Panel & Object** dialog color for the VI front panel background as shown here in the LabVIEW color picker:

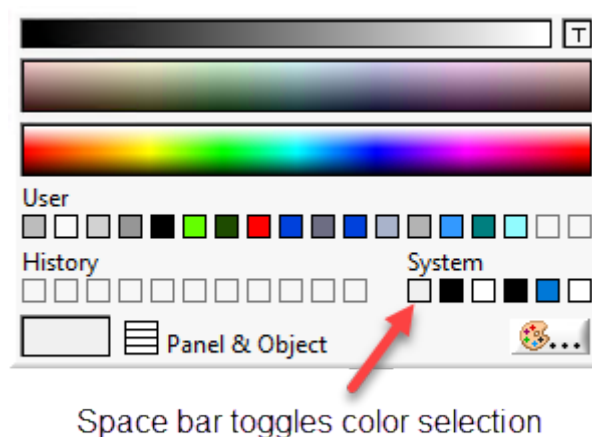


Figure 11. System colors in the LabVIEW color picker

- For dialog-style VIs use system controls where available.

6.12. Error handling



See the [What to Expect When You're Expecting an Error](#) presentation for more nuances regarding Error handling:

- Only add **error in / error out** clusters to subVIs if they can generate legitimate errors. Leave the bottom corners of the connector pane open in case the subVI needs error handling in the future.
- Use descriptive, user-friendly error messages; this makes tracing errors much easier.
- Do not show errors too early. Try to remedy them programmatically (e.g., by retrying the operation), and if the error still occurs, alert the user at an appropriate time and from the appropriate location in the program.
- Make sure your program can deal with error conditions and invalid values.
- Make test VIs that check error conditions and invalid values and operation of critical Boolean controls such as **Cancel** or **Exit**. These VIs can be part of your Unit Test suite.
- Always attempt to convert obscure errors into a more readable format.

6.13. Error logging

- Log to file and capture ALL errors for future reference.


6.14. Error codes

Use specific custom error codes to identify errors. NI reserves the following error code ranges for custom error codes:

- -8999 through -8000
- 5000 through 9999
- 500000 through 599999

It is a good idea to keep a centralized list of error codes in either a spreadsheet or database.

6.15. Block diagrams

-  Do not wire a string into a **Build Path** function unless you are programmatically generating a file or folder name via string manipulation. Always use a path constant to ensure platform portability.



You can use `..\` in your path constants instead of putting a **Strip Path** function.

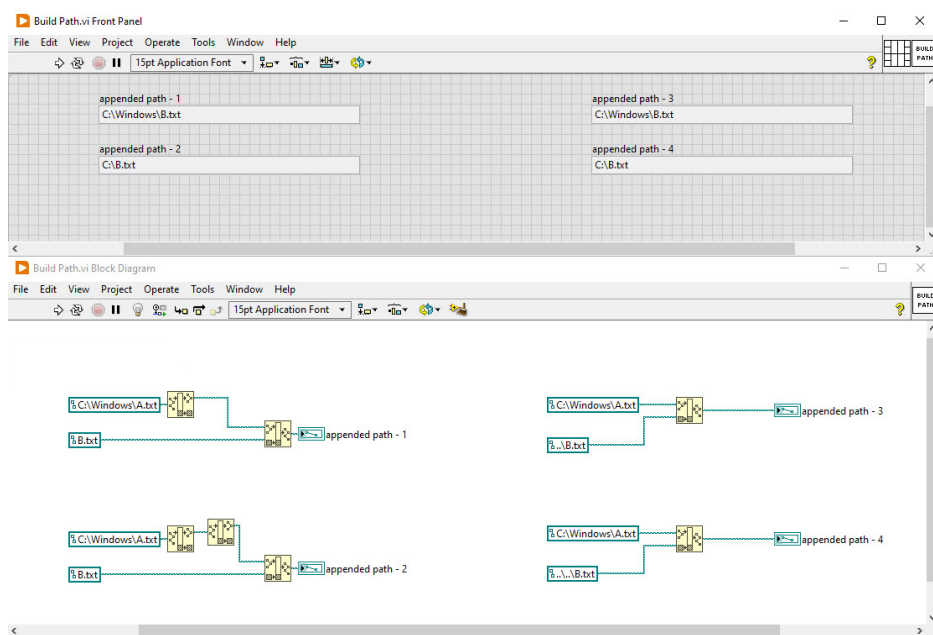




Figure 12. Different ways to build path

-  Be careful when using the **To Time Stamp** function. This function returns a value that may have an unexpected offset when used in different time zones. Similarly, be careful when using **Format Into String** to format floating point values as time stamp values. You may need to convert the floating-point value to a time zone-independent time stamp before wiring it to the **Format Into String** function. If either of these scenarios is causing time zone-related issues in your code, consider writing your own conversion subVI that utilizes the **Date/Time to Seconds** function to construct a time stamp based on a specified number of seconds.
-  Do not use the **Current VI's Path** function to build paths to files on disk in code that will eventually be built into an executable. Instead, use the **Application Directory** VI, which will give you a proper base path for code running in the development environment, or in a built executable.
- Avoid the use of the **Value Signaling** property for inter-process communication.

- Only use carriage returns in free labels to separate paragraphs. The free label must not be “Size to Text” (unless it is a single line), and instead must be sized so it will word wrap and have extra room.
- Never password protect any VIs that are part of an example, template, or sample project.
- Size the block diagram to fit the entire toolbar (i.e., enough to show the search bar), and save it slightly lower than the front panel window bar, and slightly to the right of the front panel scrollbar, so that both windows are accessible from their window bars when open.



In LabVIEW 2019 and later you can use the Arrange VI Window shortcut from Quick Drop (Press **Ctrl** + **Space**, then **Ctrl** + **F**) to automatically position the diagram in this manner.

- Do not save the block diagram maximized.
- Resize Event Data Nodes to only contain the data being used. If no data is being used, resize to one element and move to the bottom corner of the frame. In LabVIEW 2020 and later, you can right-click the Event Structure and deselect *Visible Items > Event Data Node for this Frame* to hide an unused event data node entirely.
- Disable Auto Error Handling for all VIs.
- Do not create any backward wires unless they are connected to feedback nodes.
- Do not create wires that travel under structures, or any other block diagram objects.
- Select the “View Cluster as Icon” option for all cluster constants that contain default values for the cluster contents. Create a <Cluster Name> Default—constant.vi for default values. Do not rely on the default values of the typedef. Use this Constant VI when bundling elements to create a new cluster, this will ensure the other elements do have the default values.
- Use Linked Input Tunnels for data that passes through all the frames of Case/Event Structures.
- ★ Turn-off “View as Icon” setting for all control and indicator terminals on the diagram.
- For subVIs (or top-level VIs that do not have loops), Ensure control terminals are on the far left, and indicator terminals are on the far right, of the block diagram. Ensure all control and indicator terminals reside on the top-level diagram.
- Ensure there are no calls to obsolete or deprecated VIs, properties, or methods in your code.
- Do not use Stacked Sequence Structures, even for single-frame sequences.
- If you use colors on the block diagram, use them sparingly, intentionally and stick to your system throughout the whole project!
- Instead of showing the labels on all subVIs consider referencing applicable VIs in free label comments, with the VI names in **bold**.

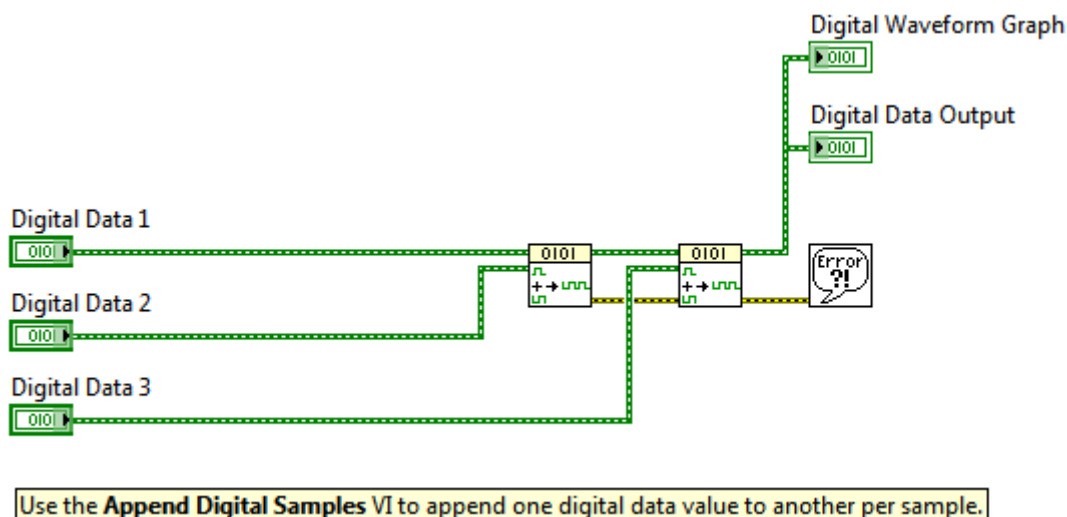


Figure 13. Comments with VI name in bold

- Think about whether your VI needs an error case structure surrounding its contents. If it is not casing out critical running code (like a high-iteration loop or a modal dialog), consider just passing the error wire through all diagram nodes without creating a case around the whole diagram.
- Avoid indiscriminate use of `Ctrl + B` or 'Remove Broken Wires'. This functional will remove ALL broken wires from the block diagram and not just those that you are looking at.
- Use **Create Constant/Control/Indicator** on terminals to ensure the creation of proper data types for subVI and function inputs.
- Avoid creating extremely large block diagrams. Limit the scrolling necessary to see the entire block diagram to one direction if at all.
- Label controls, important functions, block diagram constants, property nodes and structures.
- Align and distribute functions, terminals, and constants.

6.16. General programming style

6.16.1. Architecture

- Low Level API calls should not display dialogs when an error occurs. If the low-level API call needs to return an error, use an error constant to pass a custom error to the calling code.

6.16.2. LabVIEW environment settings

- ★ Set all VIs to separate source code from compiled code.
- ● Disable Automatic Error handling for all VIs; instead, the VI diagram should handle errors.

6.16.3. Case structures

- Do not couple the default case with other possible cases. Instead, always have the default case on its own and use it to identify non-covered states by either throwing an error or returning the equivalent of a null for data types.
- If using an enum to select cases, do not have a `Default` case, instead have a case for each enum value.

- ★ String case structures should have “Case Insensitive Match” enabled, as follows:

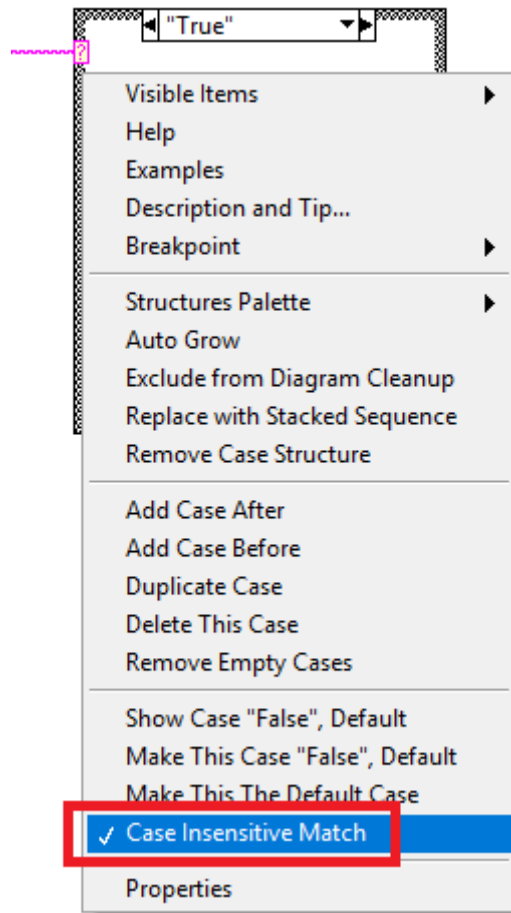


Figure 14. Case Insensitive option on case structure



With rare exception, case structures should operate on strings regardless of their casing.

- For Boolean case structures containing very simple logic, consider replacing the case structure with a `Select` function.



In performance-critical applications the case structure may execute marginally faster than the `Select` function. However, for most applications the benefits of increased readability outweigh the minor performance hit.

6.16.4. Appropriate use of data types

Use the following criteria to select a data type:

- If the same parameter is represented in other software modules, ensure that they all use the same type. Specify this up front to ensure consistency throughout the application.
- Consider what the minimum and maximum values are that any given parameter is likely to ever hold and choose the data type accordingly.

6.16.5. Data coercion



Coercion dots appear on the block diagram to indicate that LabVIEW has detected

inconsistent data types. LabVIEW will convert (cast) the data types automatically at run time.

Although in many cases, this will cause an insignificant delay during run-time as the casting takes place and the compiler makes the data conversion, it is good practice to reduce coercion dots to a minimum and ensure consistent use of data types.

- For numeric types, LabVIEW provides a Numeric Conversion palette with several functions to allow type changing.

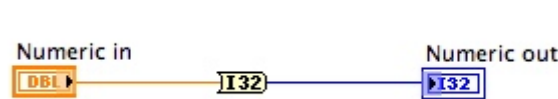


Figure 15. Numeric type conversion function

6.16.6. Event Handling

- ■ Handle all front panel events with an event structure to avoid polling front panel controls.
- Use the Event Data Node for obtaining event-related information rather than using local variables. This reduces diagram clutter but more importantly ensures strong cohesion between the event and its data.
- If there is a “Value Changed” event configured for a control, ensure the control terminal resides in the Value Changed event case. However, use the ‘New Value’ Event Data Node for any operations within the event case, this ensures the code uses the value at the time of the event regardless of what the current value is.
- Do not configure a Timeout case for an event structure unless there is reason to do so.
- ★ Never fork the Event Registration Refnum wire. Wiring the refnum to multiple event structures may result in one (but not both) of the structures handling the event.
- If your UI VI has the red ‘X’ enabled for closing the window, ensure there is a **Panel Close?** filter event that discards the event so that you can handle the cleanup and window close within the code.
- ■ Minimize the amount of code that is placed inside an event structure – it is far better technique to make use of a ‘Message Handler’ type of architecture so that all ‘functional’ and ‘handler’ code is kept separate, aiding efficiency. It also allows the handling of events to be an asynchronous activity and helps prevent duplication of code.
- ■ Use error tunnels instead of error shift registers on an event loop. If you use an error shift register, this may cause unexpected behavior if an error comes into the event structure from a previous iteration of the while loop. Use error tunnels and add conditional code to stop the while loop if an error occurs.

6.16.7. Performance Considerations

- If you use property nodes for class accessors within class member VIs, you improve code maintenance by improving the ability to find access to specific member data, but you lose performance optimizations that can be achieved when using unbundle by name instead of property nodes.
- Consider parallelizing For Loops that can be executed with parallel iterations. In situations where there are multiple nested For Loops that can be parallelized, only parallelize the outermost loop.
- Use the In Place Element Structure wherever possible to operate on array and cluster elements in place



to avoid making unnecessary copies.

- Pass individual values of arrays and clusters into subVIs instead of the entire array/cluster value to avoid unnecessary copies.
- Avoid coercion of large arrays.
- Avoid marking complex VIs as inline VIs, as this can cause substantial performance degradation when the calling VI is compiled.

Chapter 7. VI Analyzer tool

- Create a custom dictionary for VI Analyzer `Spell Check` test.



You can find more details [here](#)

- Create a VI Analyzer configuration file (`.cfg`) for each project (these files have a `.viancfg` extension in LabVIEW 2018 and later). The DQMH Consortium VI Analyzer package installs three sample VI Analyzer configuration files in `\LabVIEW Data[Company Name] CFG Templates`. Copy the appropriate `.cfg` file to your project and after each code review update the `.cfg` file to meet your project needs.

Chapter 8. DQMH

We recommend using the [DQMH®](#) when building medium to large-scale applications or when building applications that require an ‘Actor oriented design’ methodology.

- Use the DQMH Project Template as a starting point for new development / projects.
- Use the DQMH scripting tools to create new events as these ensure the User Event typedefs are updated correctly.
- Use the DQMH scripting tools to add a new module to a project.
- Use the DQMH scripting tools to rename a module.
- Use the DQMH module validation tools to validate a module against the DQMH module template and correct certain omissions.
- Consider using a helper loop in a `DQMH module Main.vi` in order to perform repeat activity (by periodically calling a module API method) instead of utilizing the timeout case of the EHL (Event Handling Loop) as this may interfere with the module’s ability to handle events in a timely manner.
- Ensure a module’s caller makes a call to `<MODULE>.lvlib:Synchronize Module Events.vi` prior to starting its own event structure as this guarantees that a module has created its own broadcast (outbound) events.

Chapter 9. Legal Information

LabVIEW is a trademark of NI (National Instruments). Neither the DQMH Consortium, nor any software programs or other goods or services offered by the DQMH Consortium, are affiliated with, endorsed by, or sponsored by National Instruments.

DQMH® is a registered trademark of DQMH Consortium, LLC.