**LabVIEW Programming and Style Guidelines**

Version 2.0

November 2013

**Purpose**

This document captures the best practices for LabVIEW programming. It is also useful as a checklist for code reviews.

These are guidelines and not hard rules; you may diverge from the guidelines if you have a good reason. Document your reasoning behind any exceptions where appropriate.

**VI Analyzer**

VI Analyzer is a tool for static analysis of VIs. You should use VI Analyzer prior to code reviews, as that will save valuable reviewer time. Green squares indicate items that can be fully or partially verified by VI Analyzer tests.

**Code Reviews**

Asking a qualified person to review your code gives you an opportunity to become a better developer. Code reviews contribute to software quality because certain classes of defects are caught earlier and cheaper than using other verification techniques.

Code reviews are most effective when they are done early and often. If you work on a project part-time, you should have code reviews at least once a week. You may want to have your code reviewed as often as every two hours.

You should use VI Analyzer and address identified issues prior to a code review. This will free the costly human reviewer to focus on aspects of code that require human attention.

**Definitions**

**Conditional** guidelines are best practices whose benefit may not be worth the time for certain projects.

**User Interface** VIs have Front Panels that a user will interact with.

**Non-User Interface** VIs have Front Panels that a user will not interact with.

**APIs** are collections of VIs with a common purpose and a consistent interface.

**Minimum Screen Resolution** is 1280x720. Front Panels and Block Diagrams using this resolution should reserve an additional 42 pixels for the Task Bar resulting in a 1280x678 resolution.

**Legend**

* White square with black border: Guideline without corresponding VI Analyzer test.
* Green square: Guideline that is fully or partially addressed by a VI Analyzer test.

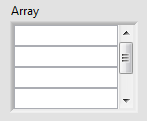
**Front Panel**

**General**

* Set controls with reasonable default values.
* Use **Size to Text** for all text one line in length. For any text longer than one line, use carriage returns only to separate paragraphs and leave empty space at the bottom to support text changes across platforms.
* Do not overlap controls and indicators.
* Leave extra space after controls set to **Size to Text** to prevent labels from overlapping objects because of font changes across platforms.

**User Interface Front Panels**

* Avoid an aesthetically unpleasing front panel. Ensure the operation of the panel is clear and that results are presented in an understandable way.
* Avoid excessive use of color. Use color logically, sparingly, and consistently, if at all.
* Include a button or other option to gracefully shut down all parts of the VI.
  + Disable the abort button
  + Consider using the system “X” button or File>>Exit option to perform the graceful shutdown instead of adding an additional exit button.
* Avoid the use of scroll bars on the dialog windows. Use scroll bars on individual controls and indicators as appropriate.
* If you use the title bar, provide a meaningful Window Title. Do not use the VI name as the Window Title.
* Hide the tool bar.
* Menus, if used, should be specific to the dialog window.
* Be consistent in using one style for controls. Use System or Silver control styles when possible.
* Use Custom Controls or “Strict Type Def.” controls to customize the appearance of a control.
  + Use Strict Type Def controls only if you have to have identical appearance of a control in multiple places.
* Use standard, consistent fonts—application, system, and dialog—throughout all user interfaces in the application.
* Configure numeric inputs with data ranges if appropriate.
* Arrange controls logically. Standard operation of an application using its UI is from Top Left to Bottom Right. Place controls accordingly in the order they are typically used.
* Consider using proximity to convey that UI elements are related.
* Consider grouping related data together into either a cluster or decoration.
  + Do not introduce clusters for UI purposes alone.
* Use proximity, alignment, decorations and UI features such as Tab Controls and subpanels to organize UI elements.
* Limit dialog use to critical user interactions.
  + All dialogs must be modal to the main application or floating.
* Consider using an array style which is more consistent with similar data structures, like listboxes. For example, remove the index control and add a scrollbar:



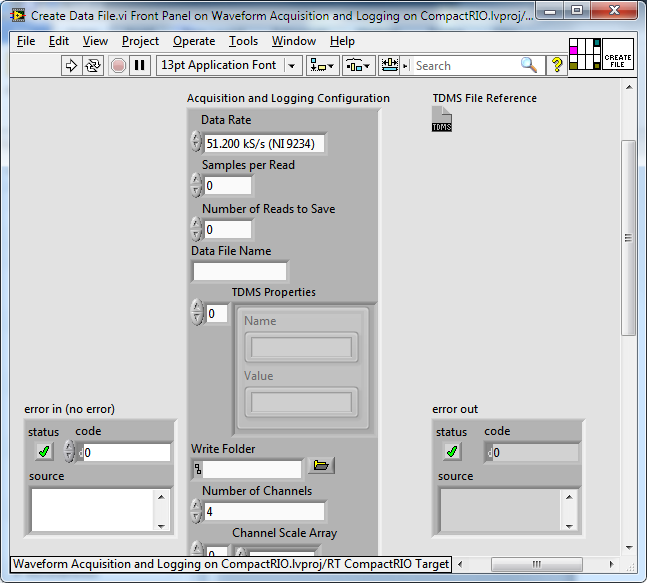
* Configure front panel to fit the monitor resolution used in the application. If no resolution is specified, configure front panel to fit within the Minimum Screen Resolution.
* Give controls and indicators a meaningful label or caption.
  + Limit caption use to those controls and indicators that require more information than the label conveys, where run-time changes to the UI are necessary, or for projects which will be localized.

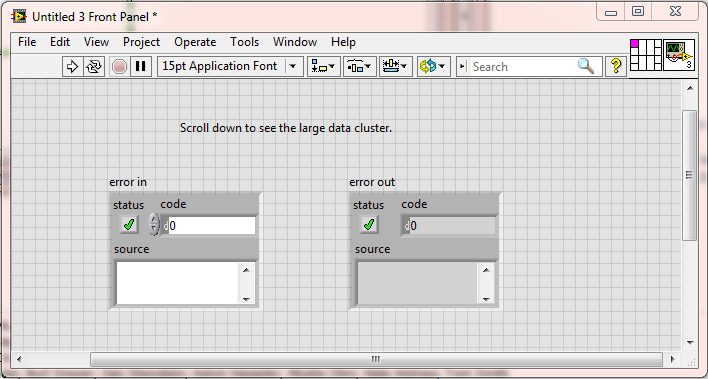
**Conditional**

* Write descriptions and create tip strips for UI front panels and controls.
* Set up appropriate keyboard shortcuts for controls, and ensure the tabbing order makes sense.

**Non-User Interface Front Panels**

* Use the default front panel background color on all VIs.
* Use transparent front panel labels.
* Place controls and indicators consistently relative to their connector pane terminals.
* Use default colors.
* Configure front panel to fit within the Minimum Screen Resolution.
* Size the front panel to fit the entire toolbar (i.e., enough to show the search bar)
* Make sure all FP fonts are the default Application Font.
* Consider grouping related input and output data into clusters. If you do this, make sure the cluster is a typedef.
  + If you have multiple VIs operating on same clusters, consider using LabVIEW classes instead of clusters.
* If your front panel has a large, “skinny” typedef cluster (width, but not height, fits in the Minimum Screen Resolution), then size the panel to fit existing regular controls, and include part of the typedef, like so:

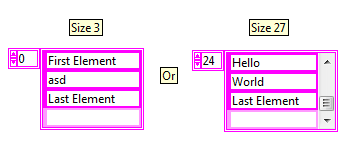


If your front panel has a large, “fat” typedef cluster (neither width nor height fits in the Minimum Screen Resolution), then include the cluster below the visible screen, along with a note indicating that the user can scroll down to see it:   


**Block Diagram Style**

**General**

* Make sure data flows from left to right (unless using feedback nodes) and wires enter from the left and exit to the right.
  + If you have a reason not to follow this guideline, provide a wire label or a floating comment for any exceptions.
* Align and distribute functions, terminals, and constants.
* Eliminate unnecessary bends/corners in wires.
* Minimize the number of wire crossings on the diagram.
* Do not place block diagram objects, such as subVIs or structures, on top of wires, and do not wire behind objects.
* Whenever possible, do not leave front panel terminals unwired in the block diagram.
* Never color diagrams.
* Use default colors for structures.
* Do not use Stacked Sequence Structures.
* Avoid the use of Flat Sequence Structures unless absolutely necessary to define execution order.
* Consider arranging parallel loops from top to bottom.
* Ensure control and indicator terminals are outside of any loops or structures unless values need to be read/updated continuously. Control terminals should be aligned at the far left and indicator terminals aligned at the far right of the block diagram.
* Make sure the subVI icon, rather than the connector pane, is visible on the block diagram.
* Display the terminal, not the icon, for front panel objects on the diagram. (Consider selecting this in your LabVIEW options dialog.) Exceptions may be made for typedefs.
* Avoid creating a string constant containing a single difficult to read value, such as a space, if the value within that constant is significant. Instead use LabVIEW’s built-in string constants.
  + For example, it is acceptable to use an empty string as a constant to specify the data type of a Queue.
* When accessing clusters, use the Bundle by Name and Unbundle by Name functions. Do not use the unnamed Bundle and Unbundle functions.
* Display large type-def cluster constants as an icon on the block diagram as long as their value is the same as the default values of the data types of the elements within the cluster type def. Otherwise use the named bundler to specify the values of the cluster. If many elements differ from the default, consider wrapping the named bundler and associated values in a subVI and inlining it.
* The first empty element in an array constant should always be displayed so that the array size is obvious. For large arrays (more than a few elements), show only a few elements and update the index such that the last element and first empty element are visible.
  + For large arrays, show the scrollbar.

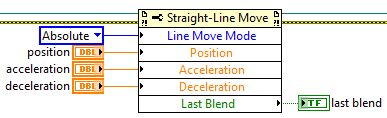


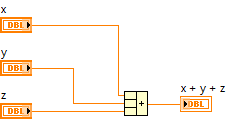
* Conditional structures (Case Structures, Event Structures, etc.) should be wide enough for the condition text to be fully visible. An exception can be made for multiple conditions in the same frame.
* Set the Name Format for Property Nodes and Invoke Nodes to Short Names to ensure the best readability of the block diagram. Long Names may sometimes be appropriate. Do not use the No Names option.
  + Apply the same reasoning for similar nodes, such as the Invoke Library Node and IP Integration Node.
* Avoid using archaic constructs such as the Code Interface Node (CIN).
* When using Block Diagram Cleanup, you must still ensure that all guidelines are followed.
* Save each VI with the most important frame of multi-frame structures (Case, Event) showing.
* Remove all unused code.

**LabVIEW Classes**

* Avoid creating excessive wire patterns and colors if you change the wire appearance of a LabVIEW class wire.
* Use dark colors on the exterior of class wires to provide good contrast against white background.
* If the class contains only refnums or static values, and the class will never contain anything that prevents safe forking of the wire, color the wire Refnum Green.

**Labels and Comments**

* All terminals should have visible and meaningful labels.
* Constants containing a non-obvious value should be documented by a meaningful label or Linked Comment.
* All labels should be transparent.
* Labels should be capitalized consistently.
* Labels should contain spaces between words.
* Avoid using duplicate labels.
* If there is a pass-through input/output pair on your example subVI you may add an ‘in’ suffix to the control, and you must add an ‘out’ suffix to the indicator. For example, if you have an input to a VI called “object”, the output should be called “object out”.
* “error in” should always contain the word “in”.
* Consider using parenthesis to document the default value and/or units of a control or indicator. Avoid using parenthesis for any other purposes.
  + In most cases, it is appropriate to not include default value as long as it is the same as the default value for the given data type.
* Use comments on the block diagram to explain the code.
* Explain why the code is doing what it is, not how. The How is usually documented by the code itself.
* Label shift registers and wires when their contents is not absolutely obvious.
* Use one of the following two styles of terminal label placement. Consider being consistent in the style chosen within a VI, a library, or a project.
  + Place terminal labels to the left of the terminal for controls and the right of the terminal for indicators, as shown here:   
     
  + Place terminal labels left-aligned above the terminal as shown here:



* The first style tends to look nice when you wire a number of controls and indicators to property nodes or VIs by eliminating wire bends. The second style tends to result in VIs that take up less horizontal space, and some people find left-aligned text easier to read.
* Only use carriage returns in free labels to separate paragraphs.
* Free labels must not be “Size to Text” (unless it is a single line), and instead must be sized so all the text is visible with extra space remaining at the bottom to account for font size changes across platforms.
* Use the standard application font in free labels on all block diagrams.
* Use the default free label color (light yellow) for all comments. Exceptions are allowed in special circumstances, such as for sample projects (LabVIEW 2012 and later).
* For subdiagram labels, use the default color (light yellow, black text) for all structures.
* For all VIs designed for consumption by a third-party (API VIs, examples, etc.), describe what the VI does in the VI Description. Consider doing this for all other VIs as well.
* Ensure proper spelling of all words on the Front Panel, Block Diagram, and in VI Properties.

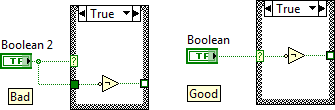
**Size and Location**

* Avoid aesthetically unpleasing block diagrams. Avoid unnecessary white space on the diagram, but don't crowd or pack your diagram too much and allow some white space to guide the eye.
* Avoid creating large block diagrams. Most VI block diagrams should fit within the Minimum Screen Resolution. When necessary, limit scrolling to one direction to see the entire block diagram.
* Size the block diagram large enough to fit the entire toolbar (i.e., enough to show the search bar)

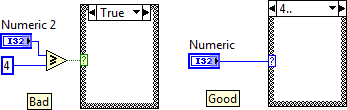
**Programming**

**General**

* Use “Type Def.s” (not “Strict Type Def.s”) to preserve the data type of common controls, especially for enumerated type controls and data structures.
  + “Common” means used in more than one place.
* Use enumerated type controls where appropriate.
  + If you are using a Boolean control for two options, consider using an enumerated type control instead to allow for future expansion of option.
  + Use an enumerated type over a ring control when possible.
  + If you must use a ring, consider making it a “Strict Type Def.” so that menu items will update. If you do make a strict type def, extend the size beyond the longest words to simplify future expansion.
* For Case Structures, do not create a separate tunnel for the wire connected to the selector terminal; wire the selector terminal inside the structure instead.



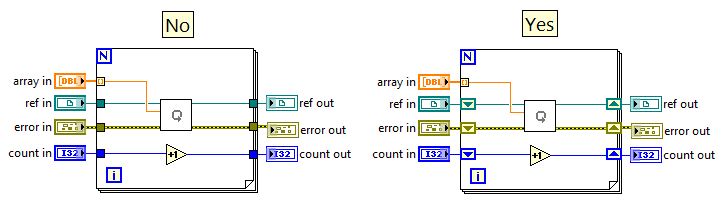
* Avoid expressions with Boolean outputs prior to Case Structures when possible; use non-Boolean selectors instead.



* Avoid the Default case in Case Structures based on enums.
* Consider name collisions when using named queues and named notifiers. Pass the reference wire in place of using named queues and notifiers wherever possible.
* If you open references such as an application, control, VI, file, etc., close the references by using the Close Reference function.
  + Close the reference in the same VI where you opened it. It is OK to use subVIs for opening and closing; if you do, pair those subVIs up. VI names and icons should reflect these pairings. For example, use pairs of words such as Open/Close or Obtain/Release.
* Use path controls/constants instead of string controls/constants to specify the location of files or directories.
* Avoid using absolute paths in VIs. Whenever possible use relative paths and define a root path once in your code or determine it dynamically at run-time based on system settings or the application location. Ensure that relative file paths are appropriately derived for Run-Time and the Development System.
* Avoid using a variable when you can use a wire to transfer data. Every local variable that reads the data makes a copy of the data. Use global and local variables only when a wire is not suitable.
* Use in-place structures when applicable to more clearly convey the movement of data.
* Consider using data value references or classes instead of Functional Global Variables whenever possible. Wires are generally easier to follow and debug than FGVs.
* Do not duplicate code. Use subVIs to encapsulate and modularize different parts of your application. Create each subVI such that it has a single clear purpose so that it is easy to reuse and to test.

**Corner Cases**

* Avoid race conditions by using appropriate synchronization techniques.
* Make sure unsupported values are handled in an appropriate manner. For example if creating an API, consider returning errors for unsupported inputs. If creating a user interface, configure the input ranges of controls.
* Ensure For Loops behave correctly if they execute zero times. Use a shift register if necessary to pass a value through the loop in this situation.
  + If you have value or a reference going through a For loop, use shift registers, not tunnels.



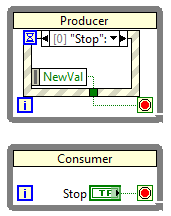
* Only use “Use Default if Unwired” on tunnels of case and event structures when confident that adding new cases or events in the future will not introduce a bug in the program. When in doubt, avoid using this feature.
* Do not use both Auto-indexing and the N terminal to govern iterations of the same For Loop. Be careful when Auto-indexing on multiple arrays of different sizes.
* Do not create clusters that contain multiple elements of the same name.
* Take special care when creating VIs for execution on multiple platforms.
  + Avoid using VIs that only execute properly on a specific platform if possible. Otherwise, consider placing platform specific functionality inside of a target-specific conditional disable structure.
  + Flag these VIs as source only for better compatibility with source control.
* Consider checking at runtime whether loops are able to run at their specified rate.
* Ensure that loops can be stopped gracefully if the user wants to shut down the application.
* When using an infinite timeout, make sure you provide a mechanism to programmatically abort the wait. Avoid using arbitrary timing delays, assuming that something relevant will happen during those delays, and the proceeding without making sure that the relevant event took place.
  + If you have to have a delay, comment how you came up with the duration.

**Error Handling**

* Every error must be appropriately handled.
  + Make a conscious decision about every place where a potential error is reported inside your code. If your decision is not obvious, comment it.
  + Employ a consistent error handling strategy. Make sure the program can deal with all error conditions and invalid values. Determine the actions for errors: catch, log, terminate, notify user, etc.
* Disable automatic error handling for all VIs.
* Do not overwrite upstream errors or warnings unintentionally.
  + When passing errors through loops, use shift registers (not tunnels) to maintain any Warnings.
  + Give upstream errors priority when merging errors unless there is a specific reason for not doing so.
* VIs that have error in and error out should use a case structure to distinguish between the error and no error conditions. Exceptions include code that should execute even on incoming error or VIs that need to be optimized for performance in the no error case.
  + Use VI documentation and/or comments to document exceptions.
* Consider creating a custom error file to contain all of your error descriptions.

**Performance**

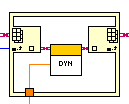
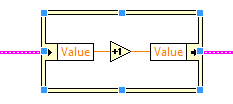
* Always keep performance in mind when designing a VI. When additional performance gains create tradeoffs with ease of development or code maintenance, choose the tradeoff most appropriate for the project.
* Avoid including code that reallocates memory inside of loops. (Build Array, Concatenate Strings)
* Avoid unnecessary type coercions.
* Avoid unnecessary data copies, especially if the data is large.
* Consider putting code that opens/closes resources (ie DAQ & File I/O) outside of loops.
* Include code to throttle the execution speed of continuously running loops.
* Use an Event Structure to monitor UI interaction where appropriate. Do not poll front panel controls when “Value Changed” events in an Event Structure would be more appropriate.
* Avoid placing time-consuming code inside of the Event Structure.
  + For example, consider using something like the producer-consumer pattern instead.
* If there is a “Value Changed” event configured for a control, the control terminal should usually reside in the Value Changed event case.

An example where it makes sense to have the control terminal outside the event structure is the stop button used in a pattern like this:  


* Resize Event Data Nodes to only contain data being used. If no data is being used, resize the node to one element and move it to a corner of the frame.

**Conditional**

* Consider using inlining and subroutine priority strategically to improve execution speeds.
* Consider parallelizing For Loops that will support it.
* Use a local variable to update a front panel object instead of the Value property.
* If performing many Front Panel updates, use the ‘Get Control Values By Index’ and ‘Set Control Values By Index’ primitives.
* Use the In Place Element Structure for Array Index/Replace Elements operations and for cluster read/modify/write operations.

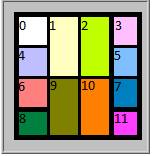
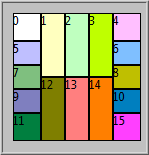
 

**SubVIs**

* Design each subVI with low coupling and high cohesion. This helps ensure each subVI is easy to understand, maintain, and test.If a group of subVIs are related, organize them in a library. Do not use LLBs.
* Create a meaningful icon for every subVI. Don't use the default icon or a blank icon. Make sure your icon will not be confused with an existing icon.

**API Design**

* Good APIs are designed such that they are:
  + Easy to learn and memorize
  + Likely to facilitate readable code
  + Hard to misuse
  + Easy to extend
  + Complete  
    (http://chaos.troll.no/~shausman/api-design/api-design.pdf)
* Use a consistent connector pane across related VIs. Wire the connector pane appropriately.
* Use the 5×2×2×2×5 (a.k.a. 5x3x3x5) or the 4x2x2x4 connector pane pattern. You must be consistent with a library, project, or development team. Avoid using connector panes with more than 16 terminals.
  + 4x2x2x4 makes sense if you work most often with core LabVIEW VIs.
  + 5x2x2x2x5 makes sense if you work most often with libraries that use this style, such as NI-DAQmx, LabVIEW FPGA VIs, etc.

**4x2x2x4 5x2x2x2x5**

* Use the left terminals for controls and right terminals for indicators. The top and bottom terminals can also be used but make sure controls are always to the left of indicators.
* Attempt to place related controls and indicators at the same relative position on the connector pane. For example, if you have a data in control at the middle left terminal, assign data out indicator to the middle right terminal.
* If a control or indicator serves the same function throughout an API, make sure that it is named consistently and located in a consistent position throughout the API’s connector panes.
* Path or references go in the top corners (terminals 0 and 3 on the 4x2x2x4 and terminals 0 and 4 on the 5x2x2x2x5 connector).
* For VIs that have a timeout input parameter, the preferred location for that parameter is terminal 6 on the 4x2x2x4 connector and terminal 9 on the 5x2x2x2x5 connector. If that location can serve a different input parameter better, the second preferred locations are terminal 1 on the 4x2x2x4 connector and terminals 1 and 2 on the 5x2x2x2x5 connector, and the third preferred locations are terminal 9 on the 4x2x2x4 connector and terminals 12 and 13 on the 5x2x2x2x5 connector.
* Error in/out on bottom left/right (terminals 8 and 11 on the 4x2x2x4 connector and terminals 11 and 15 on the 5x2x2x2x5 connector).
* The connector pane should not be fully connected to allow for future expansion.
* Set inputs to be Required for parameters that must be wired for the VI to do anything meaningful.
* Input Terminals that have good default values should be set to recommended.
* Use optional terminals only when really needed. Their main value is in hiding terminals in Context Help to reduce clutter.
* APIs should have a consistent banner and/or glyph for their icons. An icon template will help with this. (more details on icon style can be found at http://www.ni.com/white-paper/6453/en)
* Avoid using text in VI icons as they may be difficult to understand for users who do not speak your language. This is especially important for LabVIEW class public and dynamic dispatch VIs.
* Mark any VIs in the library as Private if they are not intended to be part of the API.

**Organization**

**File Naming and Organization**

* Organize all files in a manner that makes sense and can be justified.
* Give VIs meaningful names without special characters, such as backslash (\), slash (/), colon (:), and tilde (~). Ensure that VIs have names that clearly and accurately represent their function.
* Use full words with spaces for the names of all files.Use Title Case for each word in file names.
* Use standard extensions (.vi, .ctl) so the operating system can distinguish files.

Some examples:

|  |  |
| --- | --- |
| **Bad** | **Good** |
| GenericRTControl.lvproj | Generic Real-Time Control.lvproj |
| AppManagement.lvlib | Application Management.lvlib |
| ExitAllOnErr.vi | Exit All On Error.vi |
| Strings.vi | Localized Strings Global.vi |

* Use virtual folders to delineate between public and private members of a Library.
* Save custom controls with the same name as the label. For example, Alarm Boolean.ctl has the default label Alarm Boolean.
* Beware of maximum path length restrictions on Windows.

**Project Organization**

* The top-level VIs for the project must reside directly under appropriate targets.
* All subVIs should reside in libraries, classes, or virtual folders. Do not use auto-populating folders.
* Remove all unused code.
* Save the project such that it fits within the Minimum Screen Resolution.

# Localization

# The following considerations apply only to VIs and VI components that will be localized.

# General

# Display Captions instead of Labels. This may be done as part of a build step instead of during development.

# Plan for text to grow by at least 30%.

# Do not use string constants that contain text. Use Front Panel Controls instead.

# Avoid using any pictures containing text. Pictures include VI Icons.

# Considerations for Distributing Source Code

# General

* Save all VIs and projects in the upper-left region of the primary monitor.
* Save all VIs with the Block Diagram 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.
* Do not save a VI with the Front Panel or Block Diagram maximized.
* Save each VI with the most important frame of multi-frame structures (Case, Event) showing.
* Remove all breakpoints before distributing VIs.
* Consider removing the revision history of a VI so that users cannot read developer comments.
* Consider removing debugging for improved performance.

# RT-Specific Guidelines

## Timed Loops / Structures

* Whenever possible, use while loops, not timed loops, for non-deterministic tasks.
* In your documentation, use the term "deterministic" instead of "time critical" for code running in timed loops. Time Critical is commonly recognized as the VI priority, which can confuse users.
* Whenever possible, use timed loops instead of time-critical VI priority.

(http://zone.ni.com/reference/en-XX/help/370622J-01/lvrtbestpractices/rt\_priorities/)

* Serialize code within timed loops.

(http://zone.ni.com/reference/en-XX/help/370622J-01/lvrtbestpractices/rt\_priorities/)

* Consider using the inline property for VIs in timed loops to minimize subVI overhead.

(http://zone.ni.com/reference/enXX/help/371361H01/lvconcepts/vi\_execution\_speed/#SubVI\_Overhead)

* When using subVIs in timed loops that access a shared resource, consider using subroutine priority and configuring the VI to “skip call if busy”.
* Do not put controls or indicators within timed loops.

(http://zone.ni.com/reference/en-XX/help/370622J-01/lvrtbestpractices/rt\_gui\_bp/)

* With timed loops, merge inner left ear error with error input to catch any configuration errors. The same is recommended for outer right ear error and error outputs.
* Errors in time-critical code (or code essential to [TC] code) should be handled (or stop the loop and initiate a safe shutdown).
* Use select function over case structure in time-critical code, where possible. The select function provides greater determinism.

(http://zone.ni.com/reference/en-XX/help/371361H-01/glang/select/)

* All timed loops should be checked for lateness.

## Handling Data

* If file IO performance matters, establish the best chunk size for writing and reading. Note that this may vary by target.

(http://zone.ni.com/devzone/cda/tut/p/id/3746).

* Use RT FIFO functions instead of RT FIFO enabled Shared Variables for streaming data. RT FIFO enabled Shared Variables may be used for "latest update" communication.

(http://zone.ni.com/reference/en-XX/help/370622J-01/lvrtbestpractices/rt\_bp\_svars/)

* For critical data, flush file frequently. Reliance/FAT only flush once per second.
* Limit the size and quantity of log files to improve system reliability.

## Initialization / Shutdown

* Shutdown code should "un-initialize" anything initialized even if the next step is to reboot.
* Avoid writing to configuration files from within the application as this may corrupt the file preventing proper initialization on the next reboot. If you must write to a configuration file, consider developing a strategy to recover from or limit the damage of corruption.
* Consider putting code that reboots a controller in a conditional disable structure so that it can be easily disabled during debugging.

## Misc

* Use the RT palette version of Timing VIs.

(http://zone.ni.com/reference/en-XX/help/370622J-01/lvrtconcepts/timing\_control\_loops/)

* Do not use front panel objects, methods, and panel-based events in Real-Time code.

(http://zone.ni.com/reference/en-XX/help/370622J-01/lvrtbestpractices/rt\_gui\_bp/)

* Do not use “Error Cluster from Error Code”.
* Monitor the system’s processor and memory usage from development through deployment. Ensure processor usage does not stay high long enough to cause unacceptable thread starvation. Ensure memory usage does not grow continually, and that there is a plan to recover in case it does.

# FPGA-Specific Guidelines

# Programming

* Top-level VI controls and indicators are resource intensive. Keep this in mind in context of FPGA resource utilization.
* If a node you are using has an error wire output, include code to immediately handle the error. It is generally not advisable or applicable to chain VIs together using the error wire on FPGA.
* Do not write to hardware outputs inside of a Case Structure, especially when inside of a SCTL. Hardware outputs should be explicitly defined for each case and written to outside of the Case Structure
* Minimize multiple accessors to shared resources whenever possible.
* Monitor the timeout output of DMA FIFO reads and writes and respond accordingly. Also monitor the timeout output of target-scoped FIFOs unless you have a good reason not to. Possible ways to handle timeout are:
  + Latch the output, and "inform" the host that the FPGA should be restarted to ensure data integrity
  + Use a handshaking protocol to handle resending data
* Use 4-Wire handshaking anywhere appropriate, and be sure to label the wires (input valid, ready for output, output valid, ready for input).

([http://zone.ni.com/reference/en-XX/help/371599G-01/lvfpgaconcepts/fpga\_handshaking/](file:///\\nirvana\SE\public\bsnover\SE%20LabVIEW%20Style%20Guidelines\http:\\zone.ni.com\reference\en-XX\help\371599G-01\lvfpgaconcepts\fpga_handshaking\))

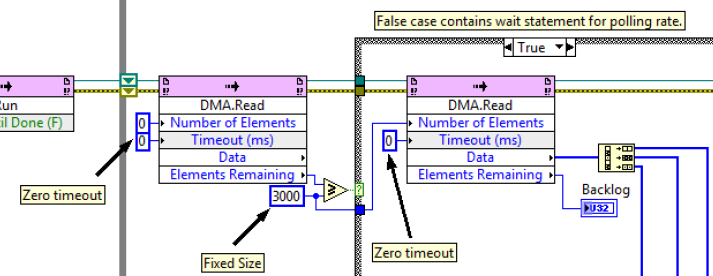
(https://decibel.ni.com/content/docs/DOC-12682)

* Whenever possible, use the FXP data-type for fractional data types unless it makes more sense to use a SGL.
* Consider converting fractional data types to SGL before passing them up to the host.

# FPGA Host Interface

* Consider calling the ‘Reset’ and ‘Run’ methods explicitly instead of automatically running the VI from the Open FPGA Reference function. This helps ensure the FPGA is in a known state before it is executed.
* When reading from DMA FIFOs:
  + Consider using the Configure method to set an appropriate host-side buffer. Avoid using large FPGA size buffers when possible to preserve block RAM.
  + Set the DMA Read timeout to zero.
  + Read a fixed-size number of elements (a multiple of the number of data channels).
  + Wait until the buffer is full before reading elements.

For Example:



* Use the Dynamic reference type when creating an API that communicates with an FPGA.