DUMPSBOSS.

Certified LabVIEW Associate Developer Examination

NI CLAD

Version Demo

Total Demo Questions: 15

Total Premium Questions: 159

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

QUESTION NO: 1

Which of the following are components of the LabVIEW waveform data type? (Choose all apply)

A. t0

B. dt

C. Y

D. A separate x value for every Y value

ANSWER: A B C

Explanation:

A waveform data type contains the Y values and timing information associated with those values. The *t0* component specifies the initial time, the *dt* component specifies the time interval between samples, and the Y component contains the measured data values.

Waveform data is useful when samples are evenly spaced in time and the timing information should travel with the data. Waveform graphs can use waveform data to display the values with the corresponding timing information. See NI documentation on the waveform data type.

QUESTION NO: 2

Which of the following statements apply to LabVIEW bookmarks? (Choose all apply)

A. They identify locations on a Block Diagram.

B. They can include a name and description.

C. They pause a VI when execution reaches the marked location.

D. They can be viewed and managed from the Bookmark Manager.

E. They enforce dataflow between nodes.

ANSWER: A B D

Explanation:

Bookmarks identify locations on a Block Diagram so that a developer can quickly navigate to areas that require attention. A bookmark can be assigned a name and a description, which is useful for recording incomplete work, review comments, or locations associated with a defect. The Bookmark Manager provides a way to view and navigate among bookmarks in a VI.

Bookmarks are development aids and do not control the execution order of the VI. They also do not pause execution like breakpoints. A breakpoint is a debugging feature that suspends a VI when execution reaches the marked location, whereas a bookmark is only a navigational annotation. See the NI LabVIEW Bookmarks documentation.

QUESTION NO: 3

Which of the following statements apply to a Conditional Disable Structure? (Choose all apply)

A. It can select code according to a target platform.

B. Only one subdiagram is enabled for a particular compilation.

C. Code in a disabled subdiagram is compiled and executed.

D. It executes each subdiagram from left to right.

E. It can use user-defined conditional symbols.

ANSWER: A B E

Explanation:

A Conditional Disable Structure selects one subdiagram to compile according to conditions such as target platform, LabVIEW version, or user-defined symbols. Only the enabled subdiagram is compiled and executed. This allows a VI to contain alternative implementations for different deployment environments without requiring each implementation to be valid for every target.

Code in a disabled subdiagram is not compiled for the current target. Therefore, a disabled subdiagram can contain functions or syntax that would otherwise be unsupported by the current target. The Conditional Disable Structure does not execute every subdiagram in order; it selects one enabled subdiagram during compilation. See the NI LabVIEW Conditional Disable Structure documentation.

QUESTION NO: 4

Which of the following are valid reasons to wire an error cluster through a sequence of SubVIs? (Choose all apply)

A. To enforce dataflow execution order between SubVIs

B. To propagate error status and source information

C. To cause an incoming error to be automatically corrected

D. To allow subsequent functions to bypass normal operation after an error

E. To guarantee that parallel loops run at the same rate

ANSWER: A B D

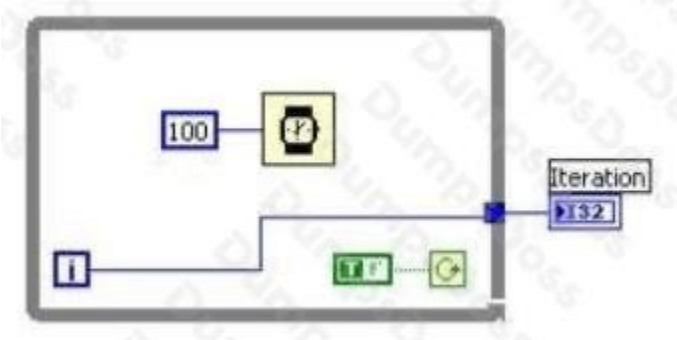
Explanation:

Wiring an error cluster through a sequence of SubVIs creates a data dependency, so a SubVI receiving the error cluster cannot execute until the preceding SubVI produces its error output. This provides a conventional way to enforce order when the functional data terminals do not otherwise establish the necessary dependency.

Error clusters also propagate error status, code, and source information through the application. Most LabVIEW functions that accept an error cluster bypass their normal operation when the incoming status indicates an error, allowing the error to reach a centralized handling location. The error wire does not itself correct an error or guarantee that independent loops execute at the same rate. See the NI LabVIEW Error Handling documentation and NI documentation on dataflow and the error cluster.

QUESTION NO: 5

Which of the following statements is true about the following block diagram?



A. The loop will execute once and the iteration terminal, , will output a value of one

B. The loop w ill execute once and the iteration terminal,, will output a value of zero

C. The loop will execute infinitely and the program will have to be aborted

D. The loop will not execute and the iteration terminal, , will return a null value

ANSWER: C

Explanation:

The loop will execute infinitely and the program will have to be aborted. A LabVIEW While Loop always executes its subdiagram at least once, then evaluates the Boolean value wired to its conditional terminal to determine whether another iteration is required. In the diagram, the condition supplied to that terminal never produces the value needed to terminate the loop. Consequently, after completing an iteration, LabVIEW continues scheduling the next iteration indefinitely rather than allowing normal dataflow to proceed beyond the loop.

The iteration terminal does not itself impose a maximum iteration count. It starts at zero for the first completed iteration and increments on each subsequent iteration, but it has no effect on stopping execution unless its value is explicitly used in logic connected to the loop's conditional terminal. Since no terminating condition is reached here, the VI remains inside the While Loop until execution is externally interrupted, such as by using the Abort Execution button or otherwise stopping the running VI. NI's LabVIEW documentation describes While Loops as repeating until their conditional terminal receives the configured stop value and notes that they execute at least once. See NI LabVIEW While Loops documentation and NI Loop Iteration Terminal documentation.

QUESTION NO: 6

Which of the following terminals controls how many times a For Loop will execute?

A.

B.

C.

D.

A. Option A

B. Option B

C. Option C

D. Option D

ANSWER: D

Explanation:

The count terminal, labeled **N** on a LabVIEW For Loop, controls how many times the loop executes. When a nonnegative integer value is wired to this terminal, LabVIEW executes the code inside the loop that number of times. For example, wiring

10 to N causes ten iterations, with the iteration terminal producing values from 0 through 9. This terminal is located at the upper-left edge of the For Loop structure. A For Loop can also be configured with a conditional terminal, which can stop the loop early when its condition is met; however, the N terminal is the terminal that establishes the requested iteration count. In the illustrated choices, the terminal identified in **Option D** is the N count terminal and therefore controls the number of loop executions. NI's LabVIEW documentation describes the For Loop count terminal and its relationship to iteration behavior in the For Loop reference. Additional details about using loop structures are available in NI's LabVIEW Structures documentation.

QUESTION NO: 7

The following figure is an example of which common type of VI architecture?

A. Multiple Case Structure VI

B. General VI

C. State Machine VI

D. Parallel Loop VI

ANSWER: C

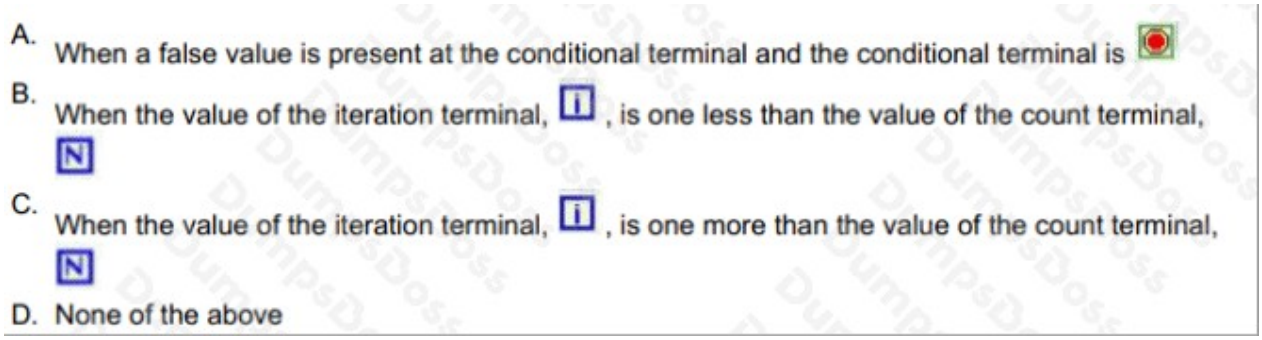
Explanation:

State Machine VI is correct. A state machine architecture organizes a LabVIEW application as a set of named operational states, with execution progressing from one state to the next according to transition logic. Typically, a While Loop repeatedly executes a Case Structure, and an enum, string, or similar state value determines which case—and therefore which action—runs during each loop iteration. Each state performs a focused task, such as initialization, acquisition, processing, waiting, error handling, or shutdown, then selects the appropriate next state.

This architecture is especially useful when program behavior must follow a defined sequence but can also branch, repeat, or respond to user input and errors. It makes complex control flow clearer because the current operation is represented explicitly as a state rather than being distributed across unrelated sections of the block diagram. NI identifies state machines as a common LabVIEW design pattern and describes their use for applications that move among distinct states based on conditions or events. For additional background, see NI's State Machine Architecture documentation and the LabVIEW design patterns overview.

QUESTION NO: 8

Under which of the following conditions does a For loop stop executing?



A. Option A

B. Option B

C. Option C

D. Option D

E. When the count terminal reaches its specified number of iterations, or earlier when an enabled “Stop if True” conditional terminal receives TRUE.

ANSWER: E

Explanation:

A LabVIEW For Loop normally completes when it has performed the number of iterations specified by its count terminal, *N*. When the loop’s optional conditional terminal is enabled and configured as “Stop if True,” the loop can also stop early as soon as that terminal receives a Boolean TRUE value. Therefore, with both terminals in use, execution ends at whichever condition occurs first: the requested iteration count is reached or the conditional stop input becomes TRUE. This behavior is useful when a task has a maximum permitted number of attempts but should terminate immediately after a measurement, search, acquisition, or calculation satisfies its completion condition. The count terminal remains an upper bound on iterations even when conditional stopping is enabled. NI’s LabVIEW documentation describes the For Loop as executing its subdiagram a set number of times and explains that an enabled conditional terminal permits early termination based on its Boolean condition. See the NI LabVIEW For Loop documentation and the NI documentation for conditional terminals.

QUESTION NO: 9

A coercion dot indicates that:

A. The data types are consistent

B. Apolymorphic operation will be performed on the data

C. A data buffer is created to handle data conversion

D. Data values are being coerced because they are out of range

ANSWER: C

Explanation:

A data buffer is created to handle data conversion is correct. In LabVIEW, a coercion dot appears on a block-diagram terminal when the data wired to that terminal has a different data type or representation from the type required by the terminal. Before the node can execute using its required input type, LabVIEW converts the incoming value. That conversion generally requires LabVIEW to create a separate data value or buffer in the required representation, such as converting an integer to a floating-point value or adapting an array element representation. The dot is therefore a useful visual indicator that a type conversion is occurring in the dataflow.

Coercion can affect both the exact numerical representation of a value and application performance, especially when it occurs repeatedly in loops or involves large arrays. Developers should inspect coercion dots during development and, where practical, wire data using the type expected by the receiving terminal. NI describes coercion dots and their role in identifying automatic data-type conversion in its LabVIEW documentation: NI LabVIEW: Coercion Dots. Related information on LabVIEW data types is available at NI LabVIEW: Data Types.

QUESTION NO: 10

Clicking on the _____ button allows you to bypass a node in the Block Diagram without single-stepping through the node.

A. Step Into

B. Step Over

C. Step Out

D. Step Through

ANSWER: B

Explanation:

Step Over is correct because it executes the currently selected node, subVI, or structure as a single debugging operation rather than entering its internal code for individual execution steps. When execution is paused at a node on the LabVIEW block diagram, clicking Step Over lets the node complete normally and advances execution to the next point at which the VI can pause. This is useful when the node's internal implementation does not need inspection but its output or subsequent dataflow does.

In LabVIEW, the execution-control buttons on the toolbar support interactive debugging of dataflow code. Step Over preserves normal node execution, including producing any outputs and observing data dependencies, while avoiding the detailed single-step path inside the selected node. This makes it particularly effective for moving past subVIs and other executable block-diagram elements efficiently while tracing a VI's broader execution behavior. NI documents these controls as part of LabVIEW's debugging tools and execution highlighting workflow. See NI's Debugging Tools Palette documentation and Debugging VIs documentation.

QUESTION NO: 11

Which of the following apply to Property Nodes? (Choose all apply)

A. Property Nodes allow attributes of front panel objects to be programmatically manipulated.

B. Property Nodes can be used to u pdate the values contained in a front panel object.

C. More than one Property Node can be used for a single front panel object.

D. Property Nodes contained in a SubVI will always cause the front panel to be loaded in memory.

ANSWER: A B C D

Explanation:

Property Nodes allow a block diagram to read or write properties of a front-panel object programmatically. These properties describe attributes and behavior such as visibility, position, size, enabled state, colors, captions, and value-related settings. Therefore, they provide a way to manipulate front-panel-object attributes while the VI runs. A Property Node can also write a control's Value property, which updates the value held by that front-panel control; its Value (Signaling) property additionally produces the control's value-change event when written.

A single front-panel object can be associated with multiple Property Nodes. For example, separate locations in the block diagram may read its current value, change its visibility, and set other properties as needed by the application's dataflow. Finally, when a SubVI contains Property Nodes that operate on its front-panel objects, LabVIEW must retain/load that front panel in memory so the referenced objects and their properties are available. This is an important consideration when designing applications intended to minimize memory use or avoid loading SubVI front panels. NI's Property Node documentation describes reading and writing object properties, and NI's guidance on front-panel loading identifies front-panel object access as a condition that can require the panel to remain in memory. See NI LabVIEW Property Nodes documentation and NI guidance on controlling when front panels load.

QUESTION NO: 12

Which of the following statements are true regarding labels and captions of Front Panel objects? (Choose all apply)

A. A label identifies the corresponding Block Diagram terminal.

B. A caption can provide descriptive text for a user.

C. Changing a caption changes the terminal name on the Block Diagram.

D. Labels can be used to identify elements in a cluster.

E. A control can have a caption but no label.

ANSWER: A B D

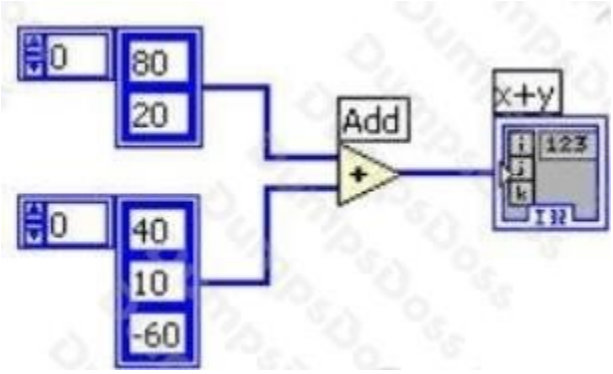
Explanation:

A label identifies a Front Panel object and is used to identify the corresponding terminal on the Block Diagram. Labels are also used by name-based functions, such as Bundle by Name and Unbundle by Name, when the object is an element of a cluster. Every control and indicator has a label, although the label can be hidden.

A caption is optional text that can be displayed with a control or indicator. Captions are intended for descriptive or user-facing text and do not determine the name of the associated Block Diagram terminal. Changing a caption does not change the object label or its terminal name. See the NI LabVIEW Labels and Captions documentation.

QUESTION NO: 13

What is the result of the following Array addition?



A. a.1-D Array of {80, 20, 40, 10, -60}

B. b.1-D Array of {120, 30, -60}

C. c.1-D Array of {120, 30}

D. d.2-D Array of {{120, 90, 20}, {60, 30, -40}}

ANSWER: C

Explanation:

The result is **1-D Array of {120, 30}**. In LabVIEW, the Add function is polymorphic: when wired with one-dimensional array inputs, it performs the numeric addition element by element, pairing values at the same index. The first corresponding values in the arrays shown produce 120, and the second corresponding values produce 30. Therefore, the output is a one-dimensional array containing those two calculated elements in their original index order.

This behavior is important when reading LabVIEW block diagrams because the array wire does not cause the Add function to concatenate values or calculate a single total. Instead, LabVIEW applies the selected numeric operation independently to corresponding array elements. The resulting wire remains an array wire, and its indicator displays the computed values as **{120, 30}**. This is an example of LabVIEW's polymorphic numeric functions, which adapt their operation to the numeric scalar, cluster, or array data types connected to their terminals. NI documents the element-by-element behavior of numeric functions and their support for array inputs in its Numeric Functions documentation and the LabVIEW Arrays documentation.

QUESTION NO: 14

Which VI memory components are ALWAYS S resident for a SubVI? (Choose all apply)

A. Data Space

B. Front Panel

C. Block Diagram

D. Code

ANSWER: A D

Explanation:

For a subVI, **Data Space** and **Code** are the VI memory components that remain resident while the VI is loaded. Data space is required to retain the run-time state associated with the VI, including values and storage needed for its controls, indicators, wires, and execution data. Keeping this information available enables LabVIEW to call the subVI and preserve the state required by its configured execution behavior.

Compiled **Code** is also resident because it contains the executable instructions LabVIEW needs to run the subVI when its caller invokes it. LabVIEW must retain the subVI's executable representation in memory so that execution can occur without requiring the development representation of the VI to be present. This distinction is important when assessing application memory use: the memory necessary for execution is separate from editor-oriented representations. NI's LabVIEW documentation describes how VI memory usage includes run-time data and compiled code, and how memory management is relevant to loaded VIs and application performance. See NI's LabVIEW Memory Usage documentation and the LabVIEW SubVIs documentation.

QUESTION NO: 15

What is an advantage of using a Strictly Typed VI ref num?

A. The data types of the target VI are known at compile time

B. The data types passed to the VI can change programmatically

C. You can flatten the data to a string to improve code performance

D. Causes dynamically loaded VIs to be loaded at the start of execution

ANSWER: A

Explanation:

The data types of the target VI are known at compile time is correct. A strictly typed VI reference is linked to a particular target VI and includes that VI's connector-pane definition. Consequently, when the reference is wired to an Invoke Node configured to call the VI, LabVIEW can identify the called VI's expected input terminals and output terminals while the caller is being developed and compiled. This allows LabVIEW to present appropriate terminals on the Invoke Node and perform compile-time type checking of the wired data.

This is especially useful for dynamically calling a known VI while preserving the clarity and safety of ordinary subVI calls. The calling diagram documents the interface it expects, and broken wires or incompatible data are identified before the application runs. If the referenced VI's connector pane changes incompatibly, dependent callers can be detected and updated as part of development rather than failing only when the dynamic call executes. NI describes strictly typed VI references as references that retain type information for a specific VI, enabling typed access through VI Server. See NI's Strictly Typed VI References documentation and the LabVIEW VI Server documentation.