DAQExpress

2025-03-20

Contents

DAQExpress 5.1 Manual

The DAQExpress 5.1 Manual contains step-by-step instructions, programming concepts, and reference information for quickly acquiring, analyzing, and presenting measurements from data acquisition devices.

Tip For interactive lessons on creating and debugging a custom application, visit the **Learning** tab in software.

Top Tasks

What do you want to do?	Where to go
Ensure software detects connected hardware.	In SystemDesigner, you can find a list of connected hardware.
Find the interactive measurement panel that's right for you.	In SystemDesigner, browse through the available measurement panels to start taking measurements with your connected devices.
Learn how to automate your measurements using G Dataflow code.	On the **Learning** tab, open interactive lessons to learn programming basics for G Dataflow.

Supported Hardware

This product supports the following hardware:

- Data Acquisition and Control
- CompactDAQ Chassis

Data Acquisition and Control

This product supports the following data acquisition and control hardware:

- Analog Input Devices and Modules

- Analog Output Devices and Modules
- Counter/Timer Devices and Modules
- Digital I/O Devices and Modules
- Digitizer Modules
- Multifunction I/O Devices
- myDAQ - Student Data Acquisition Devices
- Sound and Vibration Devices and Modules
- Strain/Bridge Input Modules
- Temperature Input Devices and Modules
- User Interface Modules

Analog Input Devices and Modules

This product supports the following C Series Current Input Modules:

- NI 9203 (Screw Terminal/Spring Terminal)
- NI 9208 (DSUB/Spring Terminal)
- NI 9227 (Screw Terminal)
- NI 9246 (Ring Lug)
- NI 9247 (Ring Lug)
- NI 9253 (Screw Terminal)

This product supports the following C Series Universal Analog Input Modules:

- NI 9218 (DSUB/LEMO)
- NI 9219 (Spring Terminal)

This product supports the following C Series Voltage and Current Input Module:

- NI 9207 (DSUB/Spring Terminal)

This product supports the following C Series Voltage Input Modules:

- NI 9201 (DSUB/Screw Terminal/Spring Terminal)
- NI 9202 (DSUB/Spring Terminal)
- NI 9205 (DSUB/Spring Terminal)
- NI 9206 (Spring Terminal)
- NI 9209 (DSUB/Spring Terminal)

- NI 9215 (BNC/Screw Terminal)
- NI 9220 (DSUB/Spring Terminal)
- NI 9221 (DSUB/Screw Terminal/Spring Terminal)
- NI 9222 (BNC/Screw Terminal)
- NI 9223 (BNC/Screw Terminal)
- NI 9224 (Spring Terminal)
- NI 9225 (Screw Terminal)
- NI 9228 (Screw Terminal)
- NI 9229 (BNC/Screw Terminal)
- NI 9238 (Screw Terminal)
- NI 9239 (BNC/Screw Terminal)
- NI 9242 (Screw Terminal)
- NI 9244 (Screw Terminal)
- NI 9251 (Mini-XLR)
- NI 9252 (DSUB/Spring Terminal)

This product supports the following C Series Frequency Input Module:

- NI 9326

Analog Output Devices and Modules

This product supports the following Analog Output Devices:

- PCI-6703
- PCI-6704
- PCI-6711
- PCI-6713
- PCI-6722
- PCI-6723
- PCI-6731
- PCI-6733
- PCIe-6738

This product supports the following C Series Current Output Modules:

- NI 9265 (Screw Terminal/Spring Terminal)
- NI 9266 (DSUB/Screw Terminal)

This product supports the following C Series Voltage Output Modules:

- NI 9260 (BNC/Mini-XLR)
- NI 9262 (DSUB)
- NI 9263 (Screw Terminal/Spring Terminal)
- NI 9264 (DSUB/Spring Terminal)
- NI 9269 (Screw Terminal)

Counter/Timer Devices and Modules

This product supports the following C Series Counter Digital Input Module:

- NI 9361 (DSUB)

This product supports the following Counter/Timer Devices:

- PCI-6601
- PCI-6602
- PCI-6624
- PCIe-6612

Digital I/O Devices and Modules

This product supports the following C Series Digital Modules:

- NI 9375 (DSUB/Spring Terminal)
- NI 9401 (DSUB)
- NI 9402 (BNC)
- NI 9403 (DSUB)
- NI 9411 (DSUB)
- NI 9421 (DSUB/Screw Terminal/Spring Terminal)
- NI 9422 (Screw Terminal)
- NI 9423 (Screw Terminal/Spring Terminal)
- NI 9425 (DSUB/Spring Terminal)
- NI 9426 (DSUB)
- NI 9435 (Screw Terminal)
- NI 9436 (Screw Terminal)

- NI 9437 (Screw Terminal/Spring Terminal)
- NI 9472 (DSUB/Screw Terminal/Spring Terminal)
- NI 9474 (Screw Terminal/Spring Terminal)
- NI 9475 (DSUB)
- NI 9476 (DSUB/Spring Terminal)
- NI 9477 (DSUB)
- NI 9478 (DSUB)

This product supports the following C Series Relay Output Modules:

- NI 9481 (Screw Terminal)
- NI 9482 (Screw Terminal/Spring Terminal)
- NI 9485 (Screw Terminal)

This product supports the following Digital I/O Devices:

- PCI-6503
- PCI-6509
- PCI-6510
- PCI-6511
- PCI-6512
- PCI-6513
- PCI-6514
- PCI-6515
- PCI-6516
- PCI-6517
- PCI-6518
- PCI-6519
- PCI-6520
- PCI-6521
- PCI-6528
- PCI-6534
- PCI-DIO-32HS
- PCI-DIO-96
- PCIe-6509
- PCIe-6535
- PCIe-6535B
- PCIe-6536

- PCIe-6536B
- PCIe-6537
- PCIe-6537B
- USB-6501 (Screw Terminal)
- USB-6509 (Mass Termination)
- USB-6525 (Screw Terminal)

Digitizer Modules

This product supports the following C Series Digitizer Module:

- NI 9775 (BNC)

Multifunction I/O Devices

This product supports the following Multifunction I/O Devices:

- PCI-6110
- PCI-6111
- PCI-6115
- PCI-6120
- PCI-6122
- PCI-6123
- PCI-6132
- PCI-6133
- PCI-6143
- PCI-6154
- PCI-6220
- PCI-6221
- PCI-6224
- PCI-6225
- PCI-6229
- PCI-6230
- PCI-6232
- PCI-6233
- PCI-6236
- PCI-6238

- PCI-6239
- PCI-6250
- PCI-6251
- PCI-6254
- PCI-6255
- PCI-6259
- PCI-6280
- PCI-6281
- PCI-6284
- PCI-6289

- PCIe-6251
- PCIe-6259
- PCIe-6251
- PCIe-6320
- PCIe-6321
- PCIe-6323
- PCIe-6341
- PCIe-6343
- PCIe-6351
- PCIe-6353
- PCIe-6361
- PCIe-6363
- PCIe-6374
- PCIe-6376

- USB-6000 (Screw Terminal)
- USB-6001 (Screw Terminal)
- USB-6002 (Screw Terminal)
- USB-6003 (Screw Terminal)
- USB-6008 (Screw Terminal)
- USB-6009 (Screw Terminal)
- USB-6210 (Screw Terminal)
- USB-6211 (Screw Terminal)
- USB-6212 (Screw Terminal/BNC/Mass Termination)
- USB-6215 (Screw Terminal)
- USB-6216 (Screw Terminal/BNC/Mass Termination)
- USB-6218 (Screw Terminal/BNC)

- USB-6221 (Screw Terminal/BNC)
- USB-6225 (Screw Terminal/BNC/Mass Termination)
- USB-6229 Screw Terminal/BNC
- USB-6251 (Screw Terminal/BNC/Mass Termination)
- USB-6255 (Screw Terminal/BNC/Mass Termination)
- USB-6259 (Screw Terminal/BNC/Mass Termination)
- USB-6281 (Screw Terminal/Mass Termination)
- USB-6289 (Screw Terminal/BNC/Mass Termination)
- USB-6341 (Screw Terminal/BNC)
- USB-6343 (Screw Terminal/BNC)
- USB-6346 (Screw Terminal/BNC)
- USB-6349 (Screw Terminal)
- USB-6351 (Screw Terminal)
- USB-6353 (Screw Terminal)
- USB-6356 (Screw Terminal)
- USB-6361 (Screw Terminal/BNC/Mass Termination)
- USB-6363 (Screw Terminal/BNC/Mass Termination)
- USB-6366 (Screw Terminal/BNC/Mass Termination)

myDAQ - Student Data Acquisition Devices

This product supports the following myDAQ - Student Data Acquisition Device:

- myDAQ

Sound and Vibration Devices and Modules

This product supports the following C Series Sound and Vibration Input Modules:

- NI 9230 (BNC/Screw Terminal)
- NI 9231 (10-32 Coaxial)
- NI 9232 (BNC/Screw Terminal)
- NI 9234 (BNC)
- NI 9250 (BNC)

This product supports the following Sound and Vibration Devices:

- PCI-4461
- PCI-4474

Strain/Bridge Input Modules

This product supports the following C Series Strain/Bridge Input Modules:

- NI 9235 (Spring Terminal)
- NI 9236 (Spring Terminal)
- NI 9237 (DSUB/RJ50)

Temperature Input Devices and Modules

This product supports the following C Series Temperature Input Modules:

- NI 9210 (Mini-TC/Spring Terminal)
- NI 9211 (Screw Terminal)
- NI 9212 (Mini-TC/Screw Terminal)
- NI 9213 (Spring Terminal)
- NI 9214 (Screw Terminal)
- NI 9216 (DSUB/Spring Terminal)
- NI 9217 (Screw Terminal/Spring Terminal)
- NI 9226 (DSUB/Spring Terminal)

This product supports the following Temperature Input Device:

- USB-TC01

User Interface Modules

This product supports the following C Series User Interface Module:

- NI 9344 (User Switch)

CompactDAQ Chassis

This product supports the following CompactDAQ Chassis:

- cDAQ-9171
- cDAQ-9174
- cDAQ-9178
- cDAQ-9179

Getting Started with DAQExpress

Getting started with DAQExpress includes an introduction to Live view in SystemDesigner, available keyboard shortcuts, tips and tricks for editing diagram code, and introduction to project documents. Click the topics in the left-hand navigation to find out more.

Keyboard Shortcuts

The following tables list keyboard shortcuts in the environment.

File Operations

Shortcut	Action
Ctrl-N	Open a new document and add it to the existing project.
Ctrl-Shift-N	Open a new project.
Ctrl-O	Open an existing project.
Ctrl-W	Close the current document.
Ctrl-S	Save the current file.
Ctrl-Shift-S	Save all open files.
Ctrl-P	Print the current document.
Alt-F4	Quit.

Basic Editing

Shortcut	Action
Ctrl-X Shift-Delete	Cut.
Ctrl-C Ctrl-Insert	Copy.
Ctrl-V Shift-Insert	Paste.
Ctrl-Z Alt-Backspace	Undo.
Ctrl-Y Alt-Shift-Backspace	Redo.
Shift-F10 Application Key	Open shortcut menu for selected item.
Delete	Delete.
Ctrl-Delete	Delete and rewire.

Selecting and Moving Objects

Shortcut	Action
Shift-Click	Select multiple objects. Add object to the current selection.
Ctrl-A	Select all objects.
Arrow keys	Move selected objects in grid-sized increments.
Shift-Arrow keys	Move selected objects four grid units.
Ctrl-Drag	Copy and drag selected object.
Ctrl-Shift-Drag	Copy selected object and move it along one axis.
Shift-Resize	Resize selected object while maintaining aspect ratio.
Ctrl-Resize	Resize selected object while maintaining center point.
Shift-Ctrl-Resize	Resize selected object while maintaining both aspect ratio and center point.
Ctrl-Drag open area	Create additional blank space along the axis you drag the mouse.
Double-click open area	Add free label or comment to panel or diagram.
Spacebar-Drag	Pan across panel or diagram.

Navigating the Environment

Shortcut	Action
Ctrl-F	Find and replace text or objects

Shortcut	Action
	within a document.
Ctrl-Shift-F	Find and replace text or objects within a project.
Enter F3 Ctrl-G	Search document for next instance of text or an object. This command is only available when in Find mode.
Shift-Enter Shift-F3 Shift-Ctrl-G	Search document for previous instance of text or an object. This command is only available when in Find mode.
Ctrl-Tab	Cycle through document tabs in the order in which they appear onscreen.
Ctrl-Shift-Tab	Cycle through document tabs in the opposite order in which they appear onscreen.
Ctrl-Shift-Spacebar	Shift focus to the application-wide search bar.
Ctrl-\	Hide and show all tabs. Hide and show all panes.

Navigating the Panel and Diagram

Editor Command	Shortcut	Action
Panel and diagram selectors	Ctrl-E	Toggle between the panel and diagram view. If the icon view is active,

Editor Command	Shortcut	Action
		switch to the panel view.
Search	Ctrl-Spacebar (Chinese keyboards) Ctrl-Alt-Spacebar	Shift focus to the palette search bar.
—	Ctrl-'	Toggle the diagram grid on/off.
—	Shift-Ctrl-;	Toggle **Smart Guides** on/off. **Smart Guides** help you align objects on the panel.
Vertical scroll bar	Mouse wheel	Scroll the document vertically.
Horizontal scroll bar	Shift-Mouse wheel	Scroll the document horizontally.
—	Tab	Shift focus from one control to another in tabbing order while the code is running.
—	Shift-Tab	Shift focus from one control to another in reverse tabbing order while the code is running.
—	Ctrl-Alt-I	Open and close the icon editor.

Debugging Commands

Editor Command	Shortcut	Action
Step In	Ctrl-Down arrow F10	Open a node and pause.
Step Over	Ctrl-Right	Execute a node without stepping into the node and pause at the next node.

Editor Command	Shortcut	Action
	arrow F11	
Step Out	Ctrl-Up arrow Shift-F11	Complete execution of the current node and pause.
Add/Remove Breakpoint	Ctrl-[	Add or remove a breakpoint on the selected node or wire to pause execution at that breakpoint.
Add/Remove Probe	Ctrl-]	Add or remove a probe on the selected wire, which allows you to view intermediate values on the selected wire as the code runs.

Help Commands

Shortcut	Action
Ctrl-H	Display the Context Help.
F1	Access additional information on ni.com.

Running Code

Editor Command	Shortcut	Action
Run	Ctrl-R	Execute this code.
Abort	Ctrl-.	Stop the code immediately, before it finishes executing.
	Ctrl-**Run** button	Recompile the code in the

Editor Command	Shortcut	Action
		current document.
	Ctrl-Shift-**Run** button	Recompile the code in all documents in memory.

Wiring

Editor Command	Shortcut	Action
Remove Broken Wires	Ctrl-B	Delete all broken wires from the diagram.
—	Esc Right-click Ctrl-Z	Delete a wire you are in the process of creating.
—	Single-click wire	Select one wire segment.
—	Double-click wire	Select a wire branch.
—	Triple-click wire	Select the entire wire.
—	Ctrl-click wire	Create a new wire branch from an existing wire.
—	Single-click while wiring	Tack down the wire segment and start a new wire segment.
—	Double-click while wiring	End the wire without connecting it to a node.
—	Tap spacebar while wiring	Switch the direction of a wire between horizontal and vertical.
Clean Up Diagram	Ctrl-U	Organize the diagram or the selected code to make it easier to understand.

Editor Command	Shortcut	Action
Clean Up Selection		
Clean Up Wire	Select **Clean Up Wire** from the shortcut menu	Route a selected wire to decrease the number of bends in the wire and avoid crossing objects on the diagram.

Editing Text

Shortcut	Action
Double-click text	Select a single word in a string.
Triple-click text	Select the entire string.
Ctrl-Right arrow Ctrl-Left arrow	Move the cursor within a string by one word in the direction of the arrow.
Home	Move the cursor to the beginning of the current line.
End	Move the cursor to the end of the current line.
Ctrl-Home	Move the cursor to the beginning of the string.
Ctrl-End	Move the cursor to the end of the string.
Esc	Cancel text entry.
Ctrl-Enter	Submit text entry.

Capturing Data

Editor Command	Shortcut	Action
Capture panel data button	Ctrl-D	Save current

Editor Command	Shortcut	Action
		data values on the panel to the Captured Data tab.

Navigating the Project Files and Captured Data Tabs

Shortcut	Action
* (Numeric keypad)	Expand everything under the selected folder.
+ (Numeric keypad)	Expand the selected folder.
- (Numeric keypad)	Collapse the selected folder.
Right arrow	Expand the selected folder if it is closed. Otherwise, this keyboard shortcut selects the first child.
Left arrow	Collapse the selected folder if it is open. Otherwise, this keyboard shortcut selects the parent.
Ctrl-up arrow Ctrl-down arrow	Scroll through the pane without changing the current selection.
Any printable key(s)	Select the item beginning with the entered letter(s).
Enter	Open the selected document.
F2	Rename the selected item.
Page Up	Move the selection to the first item in the tree.

Shortcut	Action
Home	
Page Down End	Moves the selection to the last item in the tree.

Zooming

Shortcut	Action
Ctrl-Mouse wheel	Zoom.
Ctrl-+	Zoom in.
Ctrl--	Zoom out.
Ctrl-0	Zoom to fit.
Ctrl-9	Zoom to fit the selection.

Tips and Tricks for Editing Diagram Code

The following time-saving tools can help you edit diagram code more efficiently.

Creating Diagram Code

Task	Tips and Tricks
Create constants, controls, and indicators using the shortcut menu.	You can save time when adding constants, controls, and indicators by using the shortcut menu to add the object instead of navigating the palette. Right-click any input or output of a node and select one of the Create options from the shortcut menu. For example, to create a

Task	Tips and Tricks
	constant, select the Create Constant icon.
Create a subVI from existing code.	You can use the **Create SubVI from Selection** option on the document toolbar to automatically generate a new subVI from a section of existing code. Refer to Creating a SubVI from a Section of Existing Code for more information.

Editing Diagram Code

Task	Tips and Tricks
Replace a diagram object with a similar object without using the palette.	You can replace an object on the diagram, such as a node, terminal, or constant, with a different type of the same object without navigating the palette. For example, you can replace an Add node with a Subtract node, a Numeric Terminal with a Boolean Terminal, or a True Constant with a False Constant. Right-click the object, select **Replace** from the shortcut menu, and select the desired replacement object.
Remove a structure without deleting objects in the structure.	You can remove a loop, Case Structure, or Flat Sequence from the diagram without deleting the code it contains. Right-click the border of the structure and select **Remove Name of Structure** from the shortcut menu.

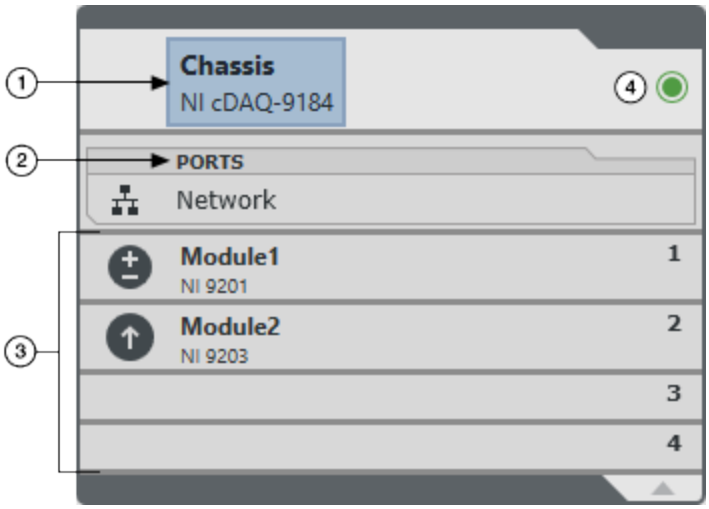
Organizing Diagram Code

Task	Tips and Tricks
Align or distribute diagram objects to improve the organization of the diagram.	On the document toolbar, click **Order**, **Align**, or **Distribute** to organize a set of selected objects.
Delete all broken wires at one time.	On the document toolbar, click **Remove Broken Wires** to delete all broken wires on the diagram.
Clean up wires.	You can automatically route a selected wire to decrease the number of bends in the wire and avoid crossing objects on the diagram. Right-click a wire and select **Clean Up Wire** from the shortcut menu.
Move a terminal label to improve diagram readability.	To change the position of a terminal label, select the terminal and update the **Label Placement** option in the Item tab.
Show a node label to improve diagram readability.	To show a node label, select the node and enable the **Show Label** option in the Item tab.

Live View: A Visual Representation of Hardware in Your System

The Live view displays all of the hardware that SystemDesigner can discover in your system. Each device on the Live diagram represents the configuration of a real or simulated device in your system.

Refer to the following image for an example of a device on the Live diagram.



1. Label and sub-label—Information about the device, such as the device name, hostname, model, product family, and serial number.
2. Ports—Connections between devices in your system.
3. Device slots—Controllers, modules, and empty slots.

> **Note** Only some devices support controllers, such as PCs and PXI chassis. If a device does support controllers, the only slot compatible with a controller is the first slot.

4. Hardware detection type—Glyph that indicates how SystemDesigner discovers the device.

Hardware Detection Type	Icon	Behavior
Auto-discovered	◉	• The device automatically appears on the Live diagram. • Changes to the live device instantly appear on the Live diagram representation of the device. • When the device is unplugged, powered down, or is no-longer reachable, it automatically disappears from the Live diagram.
Manually identified	○	• The device may automatically appear on the Live diagram, or you may need to add it. On the document toolbar, click **Add Hardware**.

Hardware Detection Type	Icon	Behavior
		• Changes to the live device may automatically appear on the Live diagram. If a change does not appear, you may need to use one of the following methods to view the change in SystemDesigner: ◦ Select the device and click **Refresh** under the **Advanced** section on the Item tab. ◦ Click **Identify Instruments** to re-scan for GPIB instruments. • When the device is removed from the system, you must manually remove it from the Live diagram.
User declared		• The device does not appear on the Live diagram until you add it manually. • Changes to the live device do not apply to the Live diagram device representation. You must manually specify changes in SystemDesigner. • To remove the device from the system, you must manually remove it from the Live diagram.

Related tasks:

- Manually Adding Hardware to the Live View of SystemDesigner

Manually Adding Hardware to the Live View of SystemDesigner

Manually add hardware devices that do not automatically appear on the Live diagram.

1. Navigate to the Live view of SystemDesigner.
2. On the document toolbar, click **Add Hardware**.
3. Follow the instructions for the appropriate device category:

Device Category	Instructions
Network Resources	Select the resource on the Add Hardware dialog box and click **Add**.
Missing SystemDesigner Support	Click **Find Support** to install required instrument drivers with NI Package Manager.

Device Category	Instructions
	Note If you install SystemDesigner support for a device, you may need to restart LabVIEW NXG before the device appears on the Live diagram.
Non-Discoverable Devices	Click **Launch NI MAX** to find and add the device to the Live diagram manually.
Device with known IP address or hostname	Click the **Add hardware by address** tab and enter the IP address or hostname of the device to which you want to connect, then click **Connect**.

Note A device category only appears when a device in your system meets the criteria for the category.

4. If the device requires login credentials, enter a valid password.
5. Click **Add** and verify that the device appears on the Live diagram.

Note If your device still doesn't appear on the Live diagram, refer to your hardware and driver manuals for troubleshooting help.

Learn More

Complete the ***Getting Started with Instrument Control*** lesson in software to learn more about setting up your devices in the Live view.

Project Documents

A ***document*** is anything that you can open, edit, and save in a project, such as a VI. You can access documents from the **Project Files** tab.

Customizing DAQExpress

Customize the DAQExpress environment by changing the mouse wheel behavior, displaying tabs in the editor, and adding guide lines to the panel and diagram.

- Customizing Mouse Wheel Behavior—Change the action the mouse wheel performs within the workspace.
- Displaying Tabs in the Editor—Access information about your project and complete project-related tasks.
- Aligning Objects on the Panel—Display guide lines to help you align and organize objects on the panel.
- Displaying the Diagram Grid—Display guide lines to help you align and organize objects on the diagram.

You can reset the workspace to the default settings.

Customizing Mouse Wheel Behavior

You can change the action the mouse wheel performs within the workspace.

1. Click **File** » **Preferences** to display the Preferences dialog box.
2. On the **Editor** tab, select the desired setting for Zoom Control.

Setting	Description
Zoom with Mouse Wheel	Scroll with the mouse wheel to zoom in and out within the workspace. To scroll vertically within the workspace, hold down the <Ctrl> key while scrolling with the mouse wheel.
Zoom with Ctrl + Mouse Wheel (Default)	Hold down the <Ctrl> key while scrolling with the mouse wheel to zoom in and out within the workspace. To scroll vertically within the workspace, scroll with the mouse wheel.

3. Click **OK**.

Displaying Tabs in the Editor

Display tabs in the editor to access different types of information about your project as well as to complete project-related tasks, such as troubleshooting or resource management.

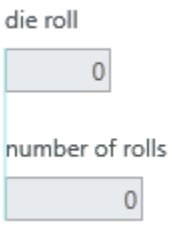
Click **View** and select the tab(s) you want to display from the pull-down menu.

Aligning Objects on the Panel

When Smart Guides are enabled, the editor displays guide lines to help you align and organize objects as you add them to the panel.

1. Click **File » Preferences** to display the Preferences dialog box.
2. On the **VI** tab, in the **Panel Editing Defaults** section, place a checkmark in the **Smart Guides** checkbox.

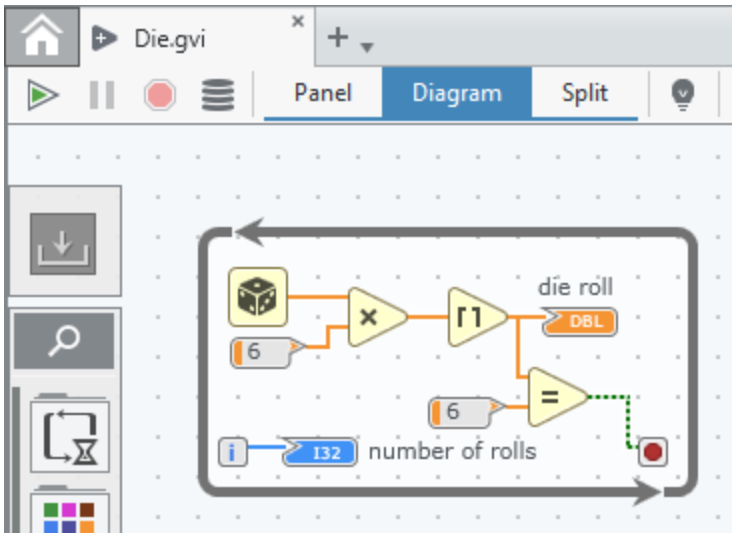
   As you add objects to the panel, the editor displays guide lines to assist with alignment, as shown in the following image.



Displaying the Diagram Grid

As you place objects on the diagram, they automatically align to a grid. Making this grid visible can help you keep the diagram organized as you add and arrange objects.

1. Click **File » Preferences** to display the Preferences dialog box.
2. On the **VI** tab, in the **Diagram Editing Defaults** section, place a checkmark in the **Always Show Grid** checkbox.



Resetting the Workspace

You can quickly restore the workspace to the default settings after you make adjustments.

Click **View** » **Reset Workspace**.

- The Navigation Pane and Configuration Pane are expanded.
- The Errors and Warnings tab is collapsed on the bottom edge of the workspace.

Creating Your First Application

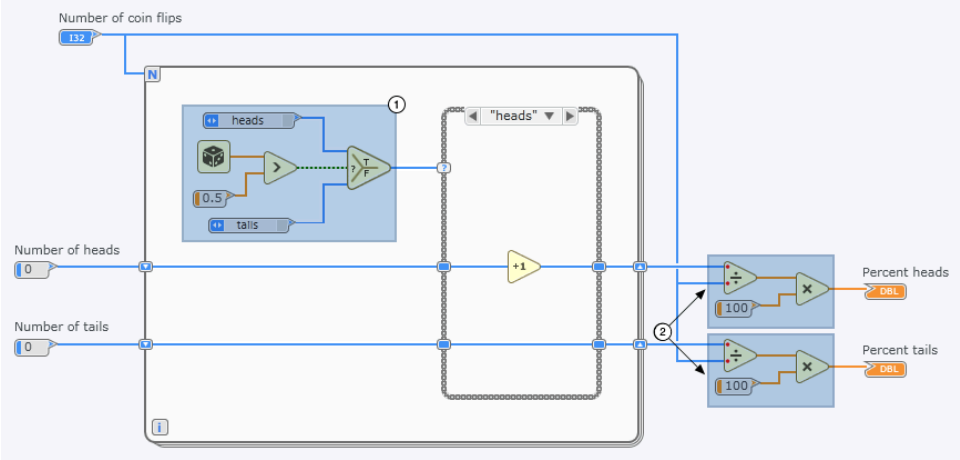
To create your first DAQExpress application, you can explore capturing and analyzing data, creating subVIs—which are analogous to functions or subroutines in other programming languages—customizing analysis functions, and analyzing data in an interactive graph. Click the topics in the left-hand navigation to find out more.

SubVIs

After you build a VI, you can call it from the diagram of another VI within the same project. A VI called from the diagram of another VI is a ***subVI***. SubVIs contain reusable code and simplify the diagrams of calling VIs. SubVIs are analogous to functions or subroutines in other programming languages.

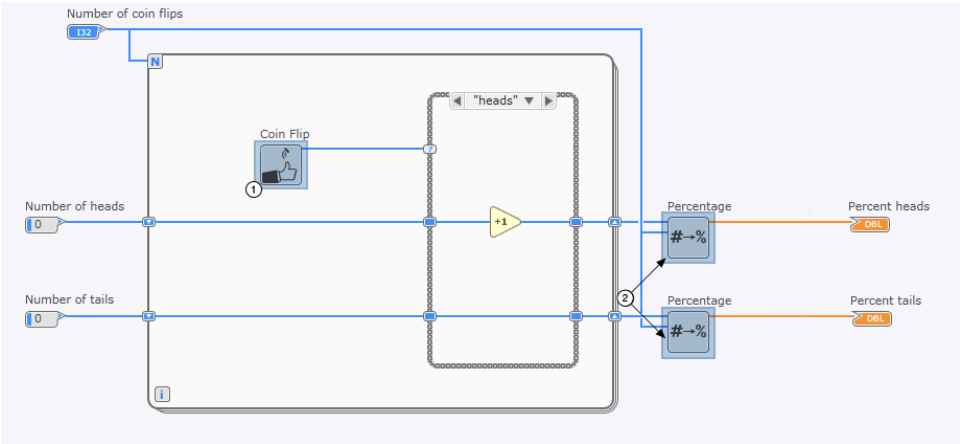
When you add an instance of a subVI to the diagram of another VI, the subVI appears as a single node, similar to a node from the palette. When data in the calling VI reaches the subVI node, the software executes the diagram of the subVI.

The following diagram shows two common situations that benefit from subVIs.



1. Users may have difficulty understanding the purpose of this section of code because it performs several operations. The more complex a diagram, the more time it takes to interpret sections of code.
2. The same code appears in two different places. Repeated code increases the likelihood of errors. For example, if you change one section of repeated code, you must remember to make the same change everywhere the code section occurs.

The following diagram shows two solutions that subVIs provide.



1. The Coin Flip subVI represents the highlighted code inside the For Loop of the previous diagram. The subVI simplifies the diagram and communicates the

purpose of the code through its label and node icon.
2. The Percentage subVI represents one of the duplicate sections of highlighted code from the previous diagram. The Percentage subVI performs the same function in two places without repeating code. If you change code in the VI called by the Percentage subVI, both instances of the subVI change.

Related tasks:

- Creating a SubVI
- Creating a SubVI from a Section of Existing Code
- Configuring an Existing VI For Use As a SubVI

Creating a SubVI

1. Right-click the diagram of an existing VI and select **Create New SubVI** to add an empty subVI node to the diagram.
   This creates a new `.gvi` file and adds it to the project. This file corresponds to the subVI node that appears on the diagram.
2. Double-click the subVI node to begin adding code to the subVI.
3. Click **Edit Icon** and add inputs and outputs to pass data in and out of the subVI. The inputs and outputs you add to the subVI icon must correspond to inputs and outputs on the diagram.

To add the new subVI as a node on other diagrams, locate the subVI in the Project Files tab and drag it to the diagram where you want to use it.

Note All instances of the subVI are automatically updated with most changes you make. However, if you remove or rearrange inputs and outputs on a subVI, wires attached to those inputs and outputs break in VIs that call the subVI. Fix the calling VIs by deleting the broken wires and rewiring the updated inputs and outputs.

Related tasks:

- Creating a SubVI from a Section of Existing Code
- Configuring an Existing VI For Use As a SubVI

Creating a SubVI from a Section of Existing Code

1. Select the section of code you want to reuse.
2. Click **Edit » Create subVI from selection** to generate a new `.gvi` file in the project. To open this file, double-click the corresponding subVI node that contains the section of code you selected. The subVI diagram and connector pane include inputs and outputs for every wire that enters and exits the section of code. The panel automatically includes the corresponding controls and indicators.

   Tip To disable automatically placing items on the panel, click **File » Preferences**, select the **VI** tab, and disable **Place items on the panel when creating subVI from selection**.

3. Rename and save the new `.gvi` file.
   Choose a name that describes what function the VI performs so that other users can clearly understand any diagram that uses the VI as a subVI.

To add the new subVI as a node on other diagrams, locate the subVI in the Project Files tab and drag it to the diagram where you want to use it.

Note All instances of the subVI are automatically updated with most changes you make. However, if you remove or rearrange inputs and outputs on a subVI, wires attached to those inputs and outputs break in VIs that call the subVI. Fix the calling VIs by deleting the broken wires and rewiring the updated inputs and outputs.

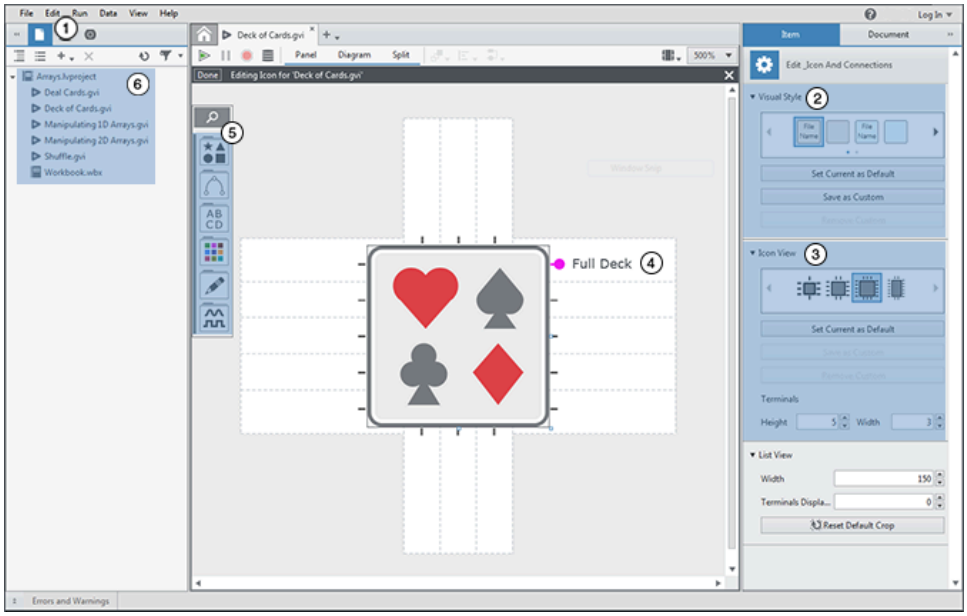
Related tasks:

- Configuring an Existing VI For Use As a SubVI

Configuring an Existing VI For Use As a SubVI

To turn an existing VI into a subVI, use the icon editor to assign input and output terminals and to customize the appearance of the subVI node.

What to Do



①	Select **Edit** » **Icon and connector pane** to display the icon editor.
②	Select an icon template. You can set a background color and choose whether to include the filename on the icon. You can also save the current design as a custom template.
③	Select a template for the terminal layout based on the number of terminals you need. **Tip** You can drag the bottom right corner of the icon to change the number of terminal selectors displayed.
④	Click a terminal selector to assign a control or indicator on the subVI panel or diagram to a specific input or output of the subVI.
⑤	Drag text and graphics from the palette to the icon to depict what the subVI does.
⑥	To use a VI as a subVI on another diagram, drag the VI from the Project Files tab to the other diagram and wire the input and output terminals.

VI Reentrancy

You can choose from three VI reentrancy options that determine how your subVI executes when it is called at more than one point in the dataflow at the same time. Reentrancy is the simultaneous execution of multiple calls for a subVI.

By default, each call to the subVI is executed one after another. This may cause your

application to execute slowly or have unexpected behavior. To allow multiple calls to a subVI to execute simultaneously, set a reentrant execution state for the subVI.

Refer to the following table to determine which VI reentrancy option is best for your subVI.

What your subVI needs	Option(s) to use
Ability to maintain state	• **Stateful** maintains state for each instance of the subVI. • **None** maintains a single state across all instances of the subVI.
Minimize wait time when multiple VIs call your subVI at the same time	• **Stateful** eliminates wait time completely. • **Stateless** results in wait time if there are more calls to the subVI than there are clones. However, the wait time is always within a certain range.
Determinism, or the same wait time for each call to the subVI	• **Stateful** guarantees the same exact wait time for each call to the subVI. This minimizes jitter in applications involving time-sensitive data.
Minimize memory usage	• **None** uses the least memory. • **Stateless** uses less memory than **Stateful** if your subVI needs the other features of reentrancy.
Minimize call overhead	• **Stateful** is the most efficient for calling a subVI many times, such as within a loop.

Related tasks:

- Choosing the Right VI Reentrancy Option for a SubVI

Choosing the Right VI Reentrancy Option for a SubVI

1. On the Document tab, in the **Behavior** section, click **Properties** to access VI

Reentrancy options for your subVI.

2. In the resulting dialog box, select the **Execution** tab.
3. Choose one of the following options:

Option	Description
None	Allocates a single data space for use by all instances of the subVI. Therefore, the software executes multiple, simultaneous calls to a subVI one at a time in an indeterminate order. Because every instance of the subVI shares the same data space, all calls to the subVI also share a single state, which preserves the values of controls and uninitialized shift registers among calls.
Stateless	Allocates one copy, or clone, of the subVI for each processor on your machine. Each call to the subVI can then access one of these shared clones instead of waiting on the main subVI. If there are not enough shared clones for each call to the subVI, additional clones are allocated on demand. Each instance of the subVI may access a different clone each time it is called.
Stateful	Allocates a separate clone for each instance of the subVI. Each instance of the subVI stores data in its own pre-allocated clone.

Related concepts:

- VI Reentrancy

Capturing and Analyzing Data

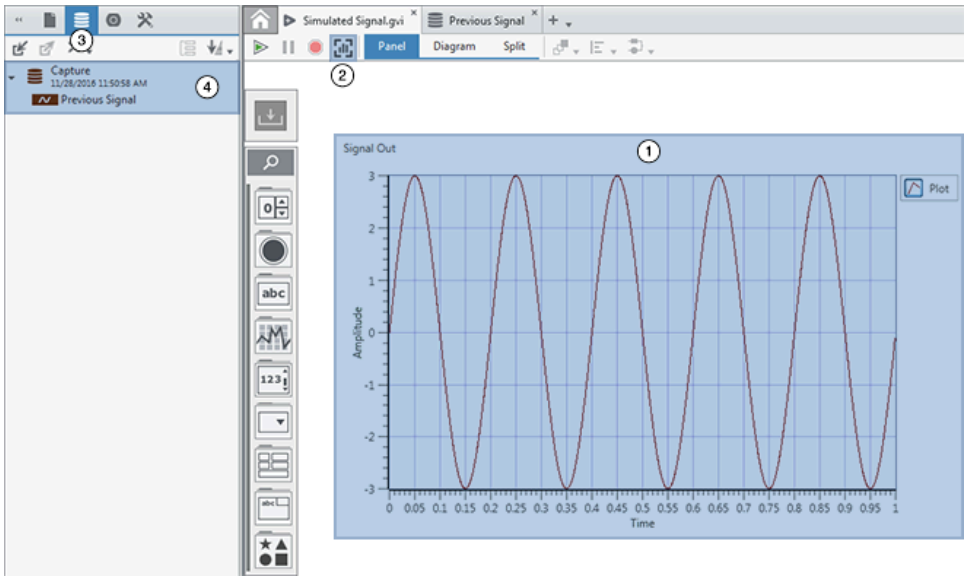
You can save and analyze the data that a device or code generates in a variety of ways. Viewing data in different formats can provide insight into trends or outliers in the data.

What to Use

- An application that generates data

What to Do

Use the editor to capture, store, and analyze data.



①	Use indicators on the panel to display data that your application generates. Right-click a control, indicator, or terminal and select **Capture data** to capture and save the data to use later.
②	Use the **Capture panel data** button to save selected data as a data item.
③	View and interact with all the data you captured in a project in the Captured Data tab. Each data item represents a piece of data that originated from a single data source at a single time.
④	You can double-click a data item to view it in the workspace, which shows the data in an interactive graph. If the data is an analog waveform of double-precision floating-point numbers or a 1D array of double-precision floating-point numbers, you can apply and customize an analysis filter by choosing an option in the Plots tab.

	Right-click a data item in the Captured Data tab and select **Export** to export data to the following file formats: • Comma-separated values file (`.csv`) • Technical Data Management Streaming file (`.tdms`) • MATLAB® formatted binary file (`.mat`)

Related tasks:

- Analyzing Data in an Interactive Graph
- Customizing Analysis Functions Using Interactive Graphs

Related information:

- Importing and Exporting MATLAB Data

Analyzing Data in an Interactive Graph

You can better understand your data by visualizing it in an interactive graph.

Viewing data in different formats can provide insight into trends or outliers in the data. For certain data types, you can also use an analysis panel to configure and apply an analysis function, capture the data that the function produces, and create a node that replicates the analysis function you configure to use in your code.

1. Capture data that your application generated using one of the following options:
   - Right-click a panel object and select **Capture data**.
   - Select **Data** » **Capture panel data**.
2. Double-click the data item in the Captured Data tab to visualize the data in the workspace.
3. Use the workspace to explore and configure the data in the graph.
4. If you want to apply an analysis function to the data in the graph, select an analysis panel on the Plots tab.

Note To use analysis panels, your data item must be an analog waveform of double-precision floating-point numbers or a 1D-array of double-precision floating-point numbers. If your data item does not meet the data type requirement, use a conversion node and capture the converted data to complete this step.

Analysis Function	Description
Amplitude Measurements	Performs peak amplitude measurement.
Curve Fit	Finds the line that best represents an input signal or input data set using a specific fitting method.
FFT Spectrum	Computes the averaged FFT spectrum of a signal.
Filter	Applies a lowpass, highpass, bandpass, or bandstop filter to a signal.
Histogram	Finds the discrete histogram of a signal.
Limit Testing	Performs limit testing by comparing the signal to upper and lower limits that you define.
Pulse and Transition Measurements	Returns various measurements of a specific pulse in a periodic waveform or an array of periodic waveforms, such as the period, pulse duration, and duty cycle.
Resample	Resamples a signal according to a specific delay and sampling interval.
Scaling and Mapping	Scales a signal based on a straight line.
Signal Correlation	Computes the auto correlation of a signal or cross correlation of two signals.
Statistics	Performs statistical calculations on a signal.
Tone Measurements	Finds tones with specific amplitude and frequency parameters.

Adjust the variables of the analysis function until it meets the requirements of your application. The graph updates in real time to show the resulting data on the same graph as the original data. You can click **Capture & view data** in the top right

corner to view only the data that the function produces with the current configuration.

> **Note** To create a node that replicates the analysis function you configured, open the **Configured Code** pane at the bottom of the analysis panel, click **Copy to Clipboard**, and then paste the node onto the diagram in a VI.

Related tasks:

- Customizing Analysis Functions Using Interactive Graphs
- Capturing and Analyzing Data

Customizing Analysis Functions Using Interactive Graphs

Use visualized data to configure filters, frequency domain transfers, and statistical analysis functions and create a customized node that you can use in your code.

1. Locate or place one of the following analysis nodes on the diagram of a VI.

Analysis Function	Description
Amplitude Measurements	Performs peak amplitude measurement.
Curve Fit	Finds the line that best represents an input signal or input data set using a specific fitting method.
FFT Spectrum	Computes the averaged FFT spectrum of a signal.
Filter	Applies a lowpass, highpass, bandpass, or bandstop filter to a signal.
Histogram	Finds the discrete histogram of a signal.
Limit Testing	Performs limit testing by comparing the signal to upper and lower limits that you define.
Pulse and Transition Measurements	Returns various measurements of a specific pulse in a periodic waveform or an array of periodic waveforms, such as the period, pulse duration, and duty cycle.

Analysis Function	Description
Resample	Resamples a signal according to a specific delay and sampling interval.
Scaling and Mapping	Scales a signal based on a straight line.
Signal Correlation	Computes the auto correlation of a signal or cross correlation of two signals.
Statistics	Performs statistical calculations on a signal.
Tone Measurements	Finds tones with specific amplitude and frequency parameters.

2. Select the node and click **Analysis panel** on the Item tab.
3. Adjust the variables of the analysis node until it meets the requirements of your application.
   a. Above the graph, choose between a sample input and any valid data item that you previously captured and saved in the project.
   b. To the right of the graph, adjust the inputs of the node.

   The graph updates in real time to show sample data and the data resulting from the analysis function. Click **Capture & view data** in the top right corner to view only the data that the function produces with the current configuration.
4. Open the **Configured Code** pane at the bottom of the analysis panel to view the customized node that corresponds to the function. To update the configuration of the node on the diagram, click **Update node**. If you want to create a new node with the current configuration, click **Copy to Clipboard** and then paste the new node onto the diagram.

Related tasks:

- Resizing Data Sets to Open in Analysis Panels
- Capturing and Analyzing Data

Resizing Data Sets to Open in Analysis Panels

If you need to open a large data set in an analysis panel, you must first trim the data to a smaller size.

Analysis panels have the following upper limits to the amount of data they can

process:

- 1 million samples per channel when you launch analysis panels from the **Interactive Analysis** section
- 9 MB total file size when you select a data set from the **Input Signal** pull-down menu of an analysis panel

Complete the following steps to resize a data set:

1. Double-click the data item to open it in the workspace and use the thumbnail view to select a subset of the data.
2. Click **Crop to Frame** to create a new capture of the subset.
   If the subset meets the upper limits criteria, you can open the subset in an analysis panel.

Warning about the Abort Button

The **Abort** button stops a program before it completes execution.

Aborting a program that uses external resources, such as external hardware or file I/O operations, might leave the resources in an unknown state and cause the program to return errors the next time you access the resources. An alternative to using the **Abort** button is to enclose code that uses external resources in a While Loop with a **Stop** button. Using the **Stop** button instead of the **Abort** button cleans up external resources before stopping execution.

Note If you are aborting a program because the code uses a Flat Sequence Structure with many frames, consider using a State Machine design pattern instead.

Collaborating on Applications

To find out more about collaborating on applications, including understanding package dependencies, sharing project and package dependencies, finding missing packages, and capturing package dependencies, click the topics in the left-hand

navigation.

Package Dependencies

In projects, a ***package dependency*** is a package installed on the development system and used in the project.

The ***Package Dependencies*** document (`.sls`) stores a list of packages a project uses so you can set up a development system with the required packages. With the Package Dependencies document, you can do the following:

- Share a project that allows a recipient to easily set up their development system with the package dependencies of the project.
- Create a project that serves as a template. Other developers can use the template project to set up their development system and begin development of the project.
- Update the list of package dependencies any time you add a package dependency to the project. Share the updated list with other developers so they can see new package dependencies you add to the project.

The Package Dependencies document is a Salt State file (`.sls`). For more information about Salt States, visit the SaltStack Documentation website and search for the SALT.STATE.PKG state module.

Related tasks:

- Sharing a Project and Including Package Dependencies

Related information:

- SaltStack Documentation
- NI Package Manager Manual

Sharing a Project and Including Package Dependencies

Use the Package Dependencies document to store a list of packages a project uses so you can set up a development system with the required packages.

1. Create a project that uses NI software, drivers, or third-party packages.
2. Capture package dependencies on the source system—Identify the packages a project requires and store a list of those packages in the Package Dependencies document.

   Note When you capture package dependencies, you must have the required packages installed on the source system and the code in the project must be in working condition.

3. Copy the project folder and save it on the target system.
4. Resolve package dependencies on the target system—Set up a development system by installing packages listed in the Package Dependencies document.

Related concepts:

- Package Dependencies

Capturing the Package Dependencies of a Project

Identify the packages a project requires and store a list of those packages in the Package Dependencies document.

1. On the Project Files tab, right-click the project and select **Capture package dependencies**.
   The Package Dependencies document opens, scans the project, and displays a list of packages the project requires.
2. (Optional) Make changes to the project, return to the Package Dependencies document, and click **Recapture dependencies**.
3. Click **File** » **Save all**.

After you capture the package dependencies of a project, copy the project folder, save it on another development system, and use the Package Dependencies document to resolve package dependencies on that system.

Related tasks:

- Sharing a Project and Including Package Dependencies

- Resolving Missing or Mismatched Package Dependencies on a Development System

Resolving Missing or Mismatched Package Dependencies on a Development System

Set up a development system by installing packages listed in the Package Dependencies document.

Before you can resolve package dependencies on a development system, copy the project folder from the system on which the project was originally developed, then save the folder on the target system.

1. Launch the project, then open the Package Dependencies document.
   When you open the Package Dependencies document, it searches the system to identify missing or mismatched packages for the project.
2. Click **Resolve**.
   The Package Dependencies document displays checkboxes next to the missing or mismatched packages. You can review the packages before installing them with NI Package Manager. By default, the Package Dependencies document selects all missing or mismatched packages.

   Note NI Package Manager does not support downgrading packages.
3. Click **Resolve selected**.
   The Package Dependencies document launches NI Package Manager.
4. Follow the prompts in NI Package Manager to install the packages you selected.
   Depending on the type of package you install, you may need to restart G Web Development Software or the development system.

Related tasks:

- Sharing a Project and Including Package Dependencies
- Capturing the Package Dependencies of a Project

Related information:

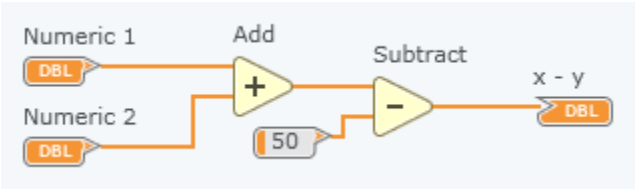
- NI Package Manager Manual

Programming in G

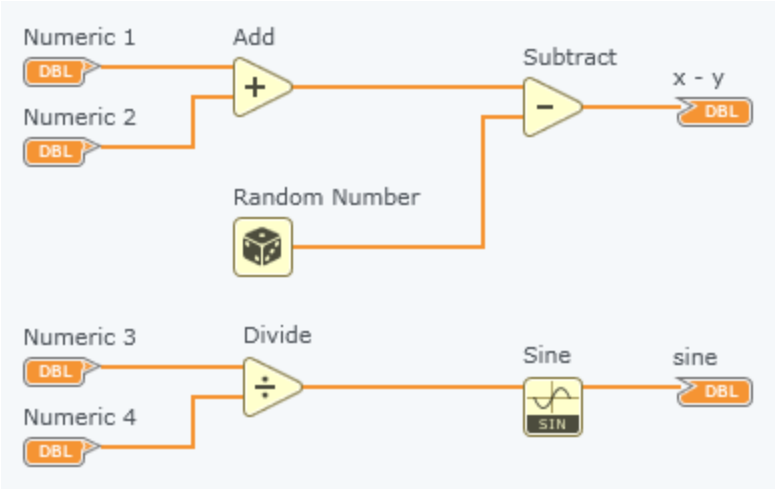
G Dataflow (G) is a graphical programming language in which nodes on a diagram execute when data is available for all required inputs.

A node in G Dataflow executes only when it receives data for all wired inputs. When a node finishes executing, it passes data along the wire to the next node in the dataflow path. The movement of data along the wires and through the nodes is called dataflow and determines the execution order of nodes on the diagram.

Because a node executes only when all of the inputs receive data, the Subtract node in the following G diagram cannot execute until Add finishes executing and passes the data to Subtract.



However, in the following G diagram, the Add, Random Number, and Divide nodes all meet the dataflow requirement of having the required input data they need to execute. Nodes not connected to each other by wires can execute in any order because the nodes do not depend on data from another node. You cannot determine which node executes first.



Because the flow of data, rather than the sequential order of commands, determines the execution order of nodes in G, you can create diagrams that have simultaneous operations. The image above illustrates operations that run simultaneously.

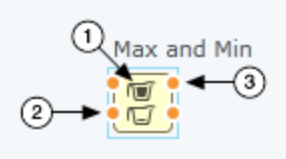
Nodes: Computational Units

Nodes are objects on the diagram that accept inputs, return outputs, and perform operations when a program runs.

They are analogous to statements, operators, functions, and subroutines in text-based programming languages.

Anatomy of a Node

Nodes have three components.



①	Icon—Illustrates the purpose of the node.
②	Input parameters—Pass data into the node. The node uses this data to perform computations.
③	Output parameters—Pass data from the node to the rest of the diagram after the node executes.

Wires: Transferring Data between Nodes

Wires transfer data between nodes on a diagram by connecting the output terminal of one node to one or more input terminals of another node.

Wires have the following properties:

- A wire has a single data source.
- A wire can carry data from its single data source to more than one node that reads the data as an input.
- The color, style, and thickness of a wire represent the type of data passing between nodes.

Troubleshooting Broken Wires

A broken wire indicates that dataflow cannot run across the wire. You cannot run code that contains broken wires. Resolve each wire to make dataflow valid and run the code.

Broken wires look like the following image.

Make sure your code meets the following requirements to resolve each wire:

- Each wire has a single data source.
- There are no unwired or overlapping tunnels when wiring through structures.

- All wires are connected to a node.
- Every data source is wired to at least one output.
  For example, you cannot wire two indicators together.
- The output and input of the same node are not wired together.
- Wire compatible data types together.
  For example, you cannot wire a Boolean output to a string input.
- Triple-click the wire to select the entire wire.
  You might see hidden wire segments that help you identify one of the causes listed above.
- Use the error list to find broken wires and an explanation.
- If you still cannot identify the cause of the broken wire, click **Remove Broken Wires** on the Document toolbar.

> **Note** **Remove Broken Wires** deletes all broken wires, even those you might not see.

Wiring Best Practices

Poor wire organization might not produce errors, but it can make the diagram difficult to read and debug or make the code appear to do things it does not do. Use these best practices as you develop your diagrams.

- Use a left-to-right and top-to-bottom layout. Although the positions of diagram elements do not determine execution order, wiring from left to right keeps the diagram organized and easy to understand. Only wires and structures determine execution order.
- Control terminal wires should exit the right side of the terminal, and indicator terminal wires should enter on the left side of the terminal.
- Wire around objects. Do not cover wires with other objects.
- Use as few bends as possible.

Wiring Shortcuts

You can use the following set of shortcuts to help you create wires more efficiently.

Editor Command	Shortcut	Action
Remove Broken Wires	Ctrl-B	Delete all broken wires from the diagram.
—	Esc Right-click Ctrl-Z	Delete a wire you are in the process of creating.
—	Single-click wire	Select one wire segment.
—	Double-click wire	Select a wire branch.
—	Triple-click wire	Select the entire wire.
—	Ctrl-click wire	Create a new wire branch from an existing wire.
—	Single-click while wiring	Tack down the wire segment and start a new wire segment.
—	Double-click while wiring	End the wire without connecting it to a node.
—	Tap spacebar while wiring	Switch the direction of a wire between horizontal and vertical.
Clean Up Diagram Clean Up Selection	Ctrl-U	Organize the diagram or the selected code to make it easier to understand.
Clean Up Wire	Select **Clean Up Wire** from the shortcut menu	Route a selected wire to decrease the number of bends in the wire and avoid crossing objects on the diagram.

Constants

A constant is a fixed piece of data that exists only on the diagram. Use a constant instead of a control when you know that a certain piece of data should always be the

same regardless of other parts of your code.

You can use the following types of constants in graphical programming:

- **Universal constant**—Represents commonly used values that are always the same, such as pi (π), the speed of light, or the character that represents pressing the Enter key.
- **User-defined constant**—Accepts user input. You define the values as you develop the code.

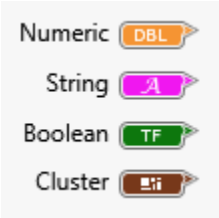
Related information:

- Constants

Terminals

Terminals transfer data between the diagram and panel, between the diagram and other nodes, or between duplicates of the same terminal on the diagram.

As the following example shows, the color and symbol of each terminal indicate the data type of the terminal.



Related concepts:

- Data Transfer between the Panel and the Diagram
- Dataflow between the Diagram and Another VI
- Dataflow between Duplicates of the Same Terminal

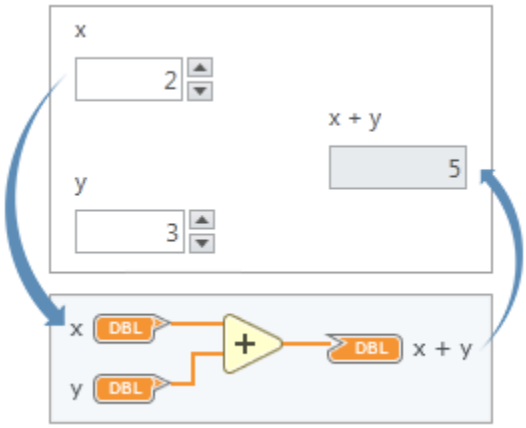
Related information:

- Data Type Reference

Data Transfer between the Panel and the Diagram

Data in a control flows from the panel to the diagram through a corresponding input terminal. Data in an output terminal flows from the diagram to the panel through a corresponding indicator.

The following image shows the flow of data between the panel and diagram of a VI.

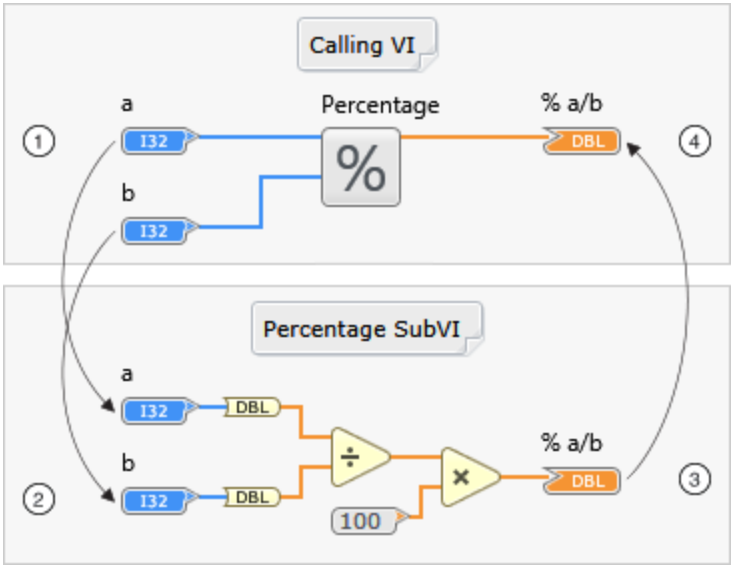


Related concepts:

- Terminals
- Dataflow between the Diagram and Another VI
- Dataflow between Duplicates of the Same Terminal

Dataflow between the Diagram and Another VI

Terminals transfer data between the diagram and other VIs via subVIs. The following image shows the flow of data between terminals of the calling VI and a subVI.



1. The calling VI passes data to the Percentage subVI through the input terminals of the subVI node.
2. Data from the calling VI flows through the input terminals on the diagram of the subVI.
3. As the subVI executes, data flows to the output terminal of the subVI diagram.
4. Data flows from the output terminal of the subVI diagram to the output terminal of the subVI node in the calling VI.

Related concepts:

- Terminals
- Data Transfer between the Panel and the Diagram
- Dataflow between Duplicates of the Same Terminal

Related information:

- SubVIs

Dataflow between Duplicates of the Same Terminal

You can use a duplicated output terminal to write to an indicator at multiple places on the diagram. You can also use a duplicated input terminal to read from a control at multiple places on the diagram.

Note Terminals are duplicates of one another if they have the exact same label. You can duplicate a terminal by right-clicking the terminal and selecting **Create Duplicate Terminal**. Copying and pasting a terminal does not create a duplicated terminal.

In the following example, the **String** indicator displays a different message depending on which While Loop is executing. This example also uses a duplicate of the **Run** button to control the execution of both While Loops.

Figure 1. Panel

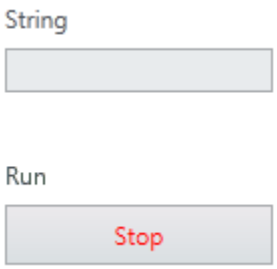
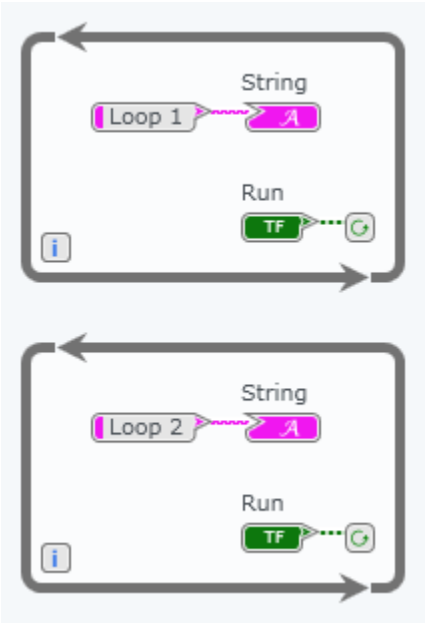


Figure 2. Diagram



Related concepts:

- Terminals
- Dataflow between the Diagram and Another VI
- Data Transfer between the Panel and the Diagram

Opening, Processing, and Closing Files

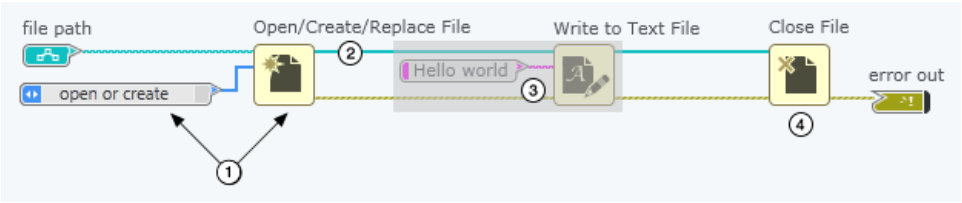
What to Use

- Open/Create/Replace File
- A read or write node from the Storage palette category
- Close File

What to Do

Create the following diagram to programmatically open, write to, and close a text file.

Customize the gray sections for your unique programming goals.



①	Open, create, or replace a file by using the Open/Create/Replace File node and specifying its precise behavior. To access the enumerated list of possible behaviors, create a constant from the `operation` input of the node.
②	After the file opens, a unique identifier called a refnum represents the file. A file refnum is required for most nodes that process files.
③	Process the file by using a file I/O node from the Storage palette category. You can interact with binary or text files. In this example, Write to Text File writes `Hello world` to the file represented by the file refnum.
④	Close the file to release the object from memory and avoid potential errors. If you do not close a file, the reference cannot be closed until the VI that opened the file finishes executing.

Troubleshooting

- Some file I/O nodes are considered high-level nodes and perform all three steps of a file I/O process—open, read/write, and close. Avoid placing high-level file I/O nodes, such as Read from Spreadsheet File and Write to Spreadsheet File, in loops because these nodes perform open and close operations each time they run.

Strategies for Improving VI Execution Speed

Factors like inefficient memory use, poorly designed panels, and too many I/O calls can negatively affect the execution speed of your VI. However, there are several strategies you can use to ensure that your VI runs more efficiently.

Strategies for Improving Execution Speed through Memory Management

Effective memory management can improve the execution speed of your VI. By minimizing memory usage, you can help alleviate slow VI execution speeds, which are affected by allocation and deallocation operations. Execution speed is indirectly affected by the allocation of more space in memory because operations slow down if they reach the memory's capacity.

The more you allocate, deallocate, or move data in memory, the more memory the VI uses. Memory can also indirectly affect execution speed because allocating more space in memory leaves less memory for other operations, causing them to slow down if they reach the memory's capacity.

When creating a VI, consider the following guidelines:

- Use consistent data types and watch for coercion dots when passing data to subVIs.
- Avoid using hierarchical data types such as arrays of clusters containing large arrays or strings.
- Reduce the use of nested subVIs inside of other VIs, as these can cause an exponential increase in the number of VI clones needed to maintain separate data spaces for each call to a subVI.

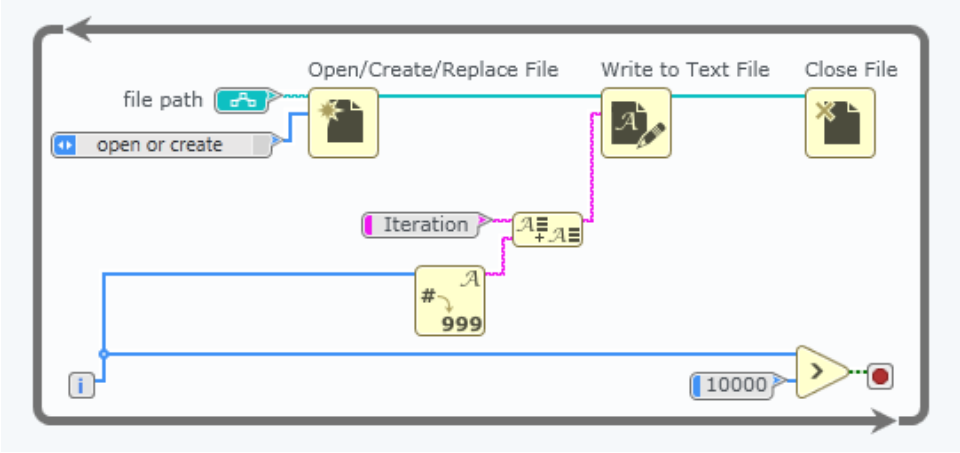
Avoid the following, which can create additional and unnecessary copies of data:

- Repeatedly resizing data—Clearing space in memory or adding and removing data from memory creates overhead and can take time.
- Overusing duplicated array and string terminals—Creating more than one corresponding object on the diagram can increase memory usage.
- Unintentionally coercing data—Coercion dots appear on nodes when data types are coerced. Data type coercion can decrease execution speed and increase memory usage, especially when you use complex data types or large arrays because they can increase memory usage.
- Displaying large arrays and strings on panels—Displaying the smallest array or string on the panel will ensure that copies of data remain small. The larger the array or string, the larger the copy of contained data.

Strategies for Improving Execution Speed by Minimizing I/O Calls

I/O calls can incur a significant amount of overhead and take more time than a computational operation because I/O calls have to wait on external resources. You can improve VI execution speed by using various strategies to troubleshoot or optimize I/O calls.

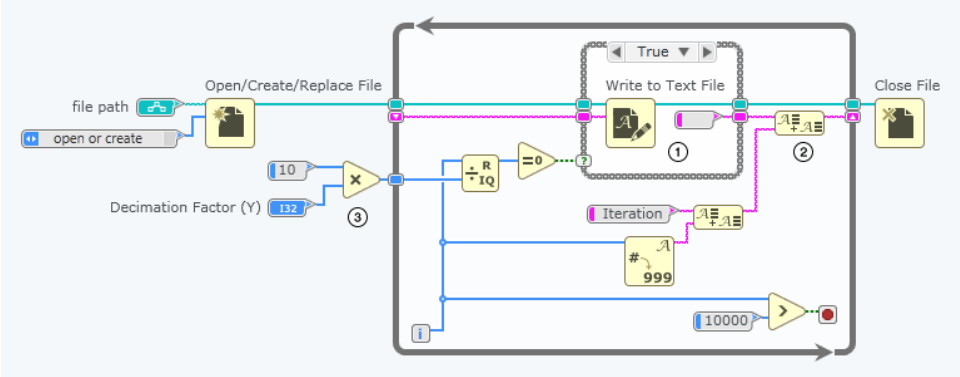
The following example diagram shows an inefficiently structured loop. The loop is inefficient because the I/O functions within the loop run multiple times unnecessarily and are structured to write a small amount of data each iteration.



When troubleshooting or optimizing your I/O calls, examine the following items to ensure your loop runs efficiently:

- I/O calls—You can reduce excessive overhead by reducing the number of I/O Write or Read calls. Limit the use of nodes within your loop that interact with external resources. Nodes like Write to Text File and TCP Write can incur the overhead of the external resource they interact with, like a network card, an OS, or file system.
- Data transfer quantities—Instead of making multiple I/O calls with smaller quantities of data, structure the VI to transfer large amounts of data with each call. Consolidate data into large chunks by using arrays and concatenated strings whenever possible.
- Unnecessary nodes outside of loops—Avoid putting a node in a loop if the node produces the same value for every iteration. Instead, move the node out of the loop and pass the result into the loop. Also, keep nodes that open or close references such as TCP Open Connection and Close File outside of loops to avoid repetitive overhead delays when interacting with external resources.

The following example diagram shows the loop restructured for efficiency:



1. The loop is efficient because it minimizes the number of I/O calls.
2. The loop is structured to transfer significant amounts of data with each call.
3. The loop does not contain unnecessary computations.

Guidelines for Designing Efficient Panels

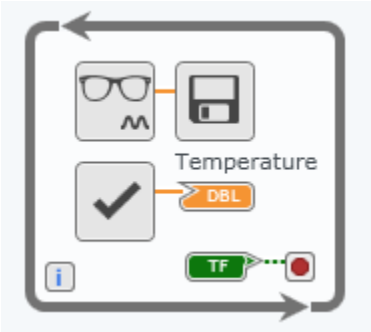
Optimizing your panel can improve the execution speed of your VI. Design and configure your panel according to the following guidelines:

- Remove unnecessary panel objects and avoid adding objects to the panel that aren't necessary for controlling input to the diagram or observing output data.
- Avoid transparent and overlapped panel objects unless necessary.
- When using charts, reducing Chart History Length in the Configuration pane will

set the maximum number of data points drawn to your chart.
- When using graphs and charts, turn off auto-scaling, axis labels, anti-aliased line drawing, and axis grids to prevent drawing more elements. A graph with fewer elements takes less time to redraw every time the graph updates.

Using Parallel Loops to Increase VI Execution Speed

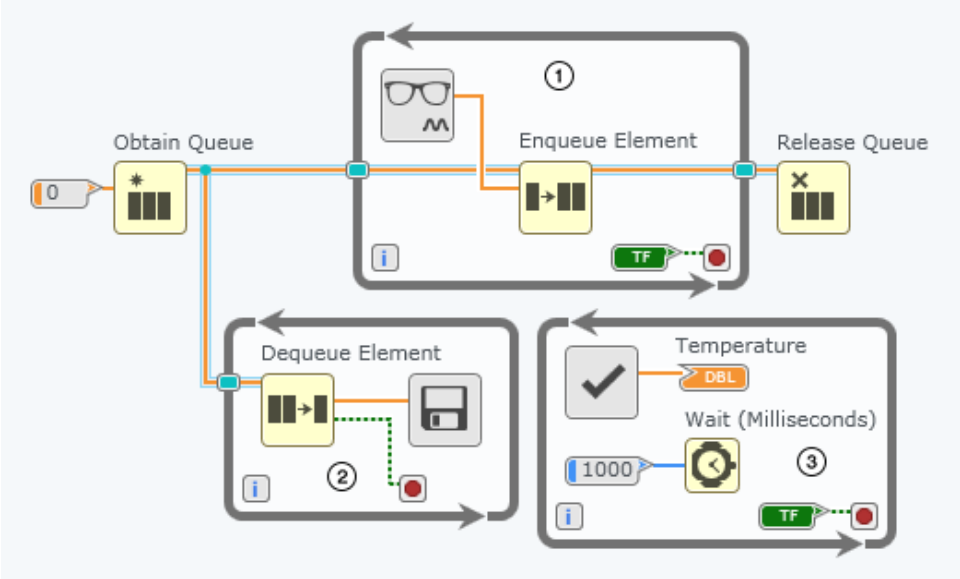
When multiple loops run in parallel, the VI switches between them periodically. Parallel loops can help decrease execution time by simultaneously running independent tasks within individual loops. In the following example diagram, a single loop contains multiple independent tasks. This loop is inefficient because execution speeds can vary due to the lack of loop timing. In this example, the DAQ operation must wait for the slower I/O function and a status check to complete each iteration.



Since all parallel loops share the same processor resources, consider the following practices to make sure other loops do not interrupt the timing or execution of your important tasks:

- Important Operations—Because you want this type of operation to execute as frequently as possible, do not add timing nodes inside the loop to delay its execution.
- Executions Dependent on Dequeue Element node—The Dequeue Element node can halt the execution of a loop until an element is ready in the queue. This node allows more important loops to use more of the processor's time.
- Low-Importance Operations—Because this type of operation is the least important, you can use a Wait node to run the operation less often and yield execution time to the other loops.

The following is an example of a recreated VI using the considerations:



1. The Read DAQ loop reads data from a DAQ device and sends the data to a queue. Do not add timing nodes because you want this type of operation to execute as frequently as possible.
2. The Write to File loop can execute more slowly as it is logging data points for later analysis. The execution of this loop can be dependent on elements being ready in the queue because it does not need to execute as frequently.
3. The Check Temperature loop displays temperature data to the user and uses a Wait node to run less often and yield execution time to the other loops because it is the least important.

Repeating Operations

When building an application, you may need to repeat code a set number of times or until a specified condition is met. For example, you may have a section of code that you know you want to run three times. You may want your code to keep running until a user clicks a stop button. You can use loops to do this.

A ***loop*** is a programming element that executes the same code multiple times, or iterations.

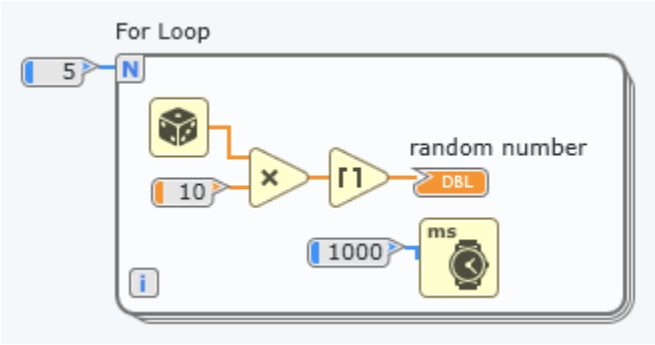
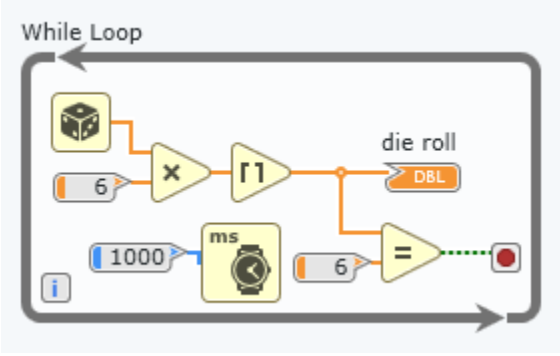
Related tasks:

- Repeating Operations until a Condition Occurs

- Repeating Operations a Set Number of Times
- Repeating Operations Once for Every Element in an Array

Types of Loops

You can use the following loops to repeat code in G Dataflow: a For Loop and a While Loop.

Loop	Behavior	Example	Diagram
For Loop	Repeats the code inside the loop for a set number of iterations.	Generate five random numbers between 1 and 10.	
While Loop	Repeats the code inside the loop until a specified condition is met.	Continue rolling a die until the value of the die is 6.	

Related concepts:

- Loop Timing

Related information:

- While Loop
- For Loop

Repeating Operations until a Condition Occurs

In some situations, you might know that you want to repeat an operation, but you do not know exactly how many times you want to repeat that operation. Instead, you know only that the operation should repeat until a certain condition occurs. For example, you might have a program you want to run repeatedly until a user clicks a stop button or until the code inside the loop produces a particular value.

Use a While Loop to repeat code until a specified condition is met. A While Loop behaves similarly to a `do while` loop in other programming languages.

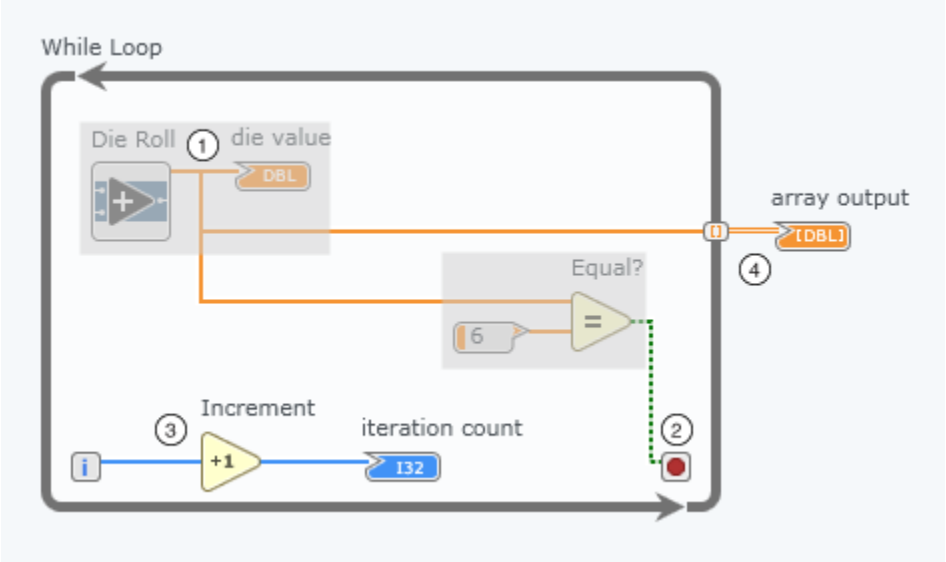
What to Use

While Loop

What to Do

Create the following diagram to repeat an operation until a condition occurs.

Customize the gray sections for your unique programming goals.



①	Place the code you want to repeat on the subdiagram of a While Loop.

②	To specify the condition under which the loop stops, create code that produces a True Boolean value when the desired stop condition occurs and wire that Boolean value to the condition terminal. By default, the condition terminal is configured to **Stop if True**. To stop the loop for a False Boolean value instead of a True value, right-click the condition terminal and select **Continue if True**. You can also specify when the loop stops by wiring an error cluster to the condition terminal. In this situation, the Boolean value of the `status` of the error is used to determine when the loop stops.
③	If you want to know how many loop iterations executed before the stop condition occurred, use the iteration terminal to return the current loop iteration count. The iteration terminal is zero-indexed, meaning it ranges from 0 to n -1. Use Increment to obtain the true number of loop iterations. **Note** A While Loop always executes at least one time.
④	If you want to collect the result of each loop iteration in an array, use an auto-indexing output tunnel to pass values out of the loop. To enable auto-indexing for an output tunnel, right-click the tunnel and select **Append Mode** » **Auto Index Values**. If you do not enable auto-indexing, the loop returns only the value from the last loop iteration.

Troubleshooting

If the execution speed of the loop is too fast, place a Wait node on the subdiagram of

the loop to specify a wait period.

Related concepts:

- Loop Timing

Related tasks:

- Repeating Operations
- Repeating Operations a Set Number of Times
- Repeating Operations Once for Every Element in an Array

Repeating Operations a Set Number of Times

Instead of creating the same code on a diagram multiple times, you can write code in a single location and use a For Loop to programmatically repeat it. For example, you might want to read a specific number of measurement samples from a device. You can place the code that performs the measurement operation on the subdiagram of a For Loop and configure the loop to repeat the operation as many times as your application requires.

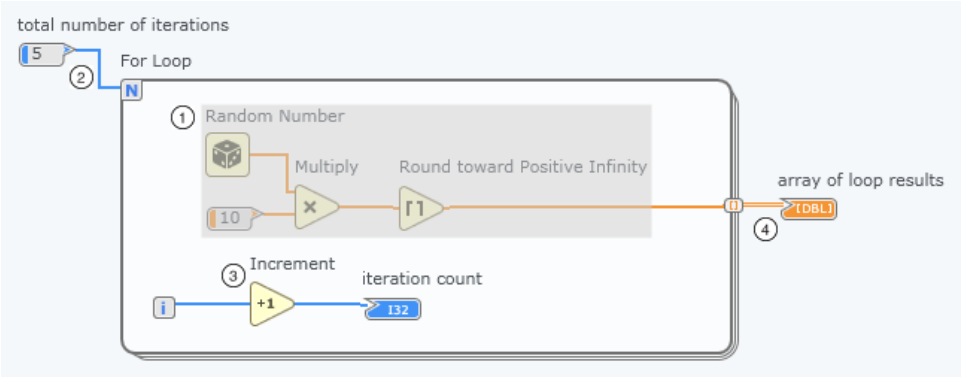
What to Use

- For Loop
- Count terminal of the For Loop

What to Do

Create the following diagram to repeat an operation a set number of times.

Customize the gray sections for your unique programming goals.



①	Place the code you want to repeat on the subdiagram of a For Loop.
②	Specify how many times the For Loop repeats its subdiagram by wiring the desired value to the count terminal. **Tip** If you want the loop to execute once for each element in an array, wire the array to the loop using an auto-indexing tunnel instead of wiring a value to the count terminal.
③	If you want to know which loop iteration is currently executing, use the iteration terminal to return the current loop iteration count. The iteration terminal is zero-indexed, meaning it ranges from 0 to n -1. Use Increment to obtain the true number of loop iterations.
④	If you want to collect the result of each loop iteration in an array, use an auto-indexing output tunnel to pass values out of the loop. To enable auto-indexing for an output tunnel, right-click the tunnel and select **Append Mode** » **Auto**

	Index Values. If you do not enable auto-indexing, the loop returns only the value from the last loop iteration.

Troubleshooting

- If the For Loop unexpectedly fails to execute, verify that the value wired to the count terminal is greater than zero.
- If the execution speed of the loop is too fast, place a Wait node on the subdiagram of the loop to specify a wait period.

Related concepts:

- Loop Timing

Related tasks:

- Repeating Operations
- Repeating Operations Once for Every Element in an Array
- Repeating Operations until a Condition Occurs

Repeating Operations Once for Every Element in an Array

While working with an array of data, you might want to access individual elements within the array. Although you can use a combination of Array nodes to accomplish this task, the For Loop includes an auto-indexing tunnel that you can use to access the individual elements with minimal additional code.

Use a For Loop with an auto-indexing input tunnel to process one element of an array during each iteration of the loop. A For Loop with an auto-indexing input tunnel behaves similarly to a `for each` loop in other programming languages.

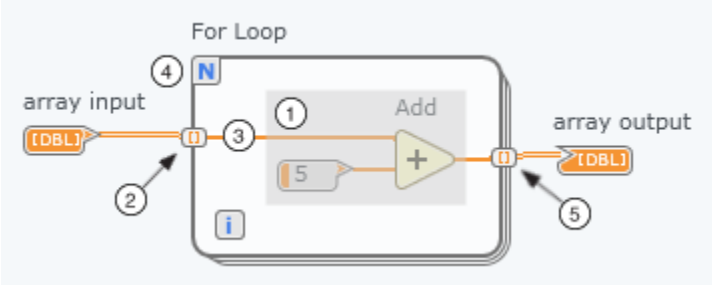
What to Use

For Loop with an auto-indexing input tunnel

What to Do

Create the following diagram to repeat an operation once for every element in an array.

Customize the gray sections for your unique programming goals.



①	Place the code you want to repeat on the subdiagram of a For Loop.
②	When you wire an array to a For Loop, the For Loop processes one element of the array at a time. This configuration, known as auto-indexing, occurs by default. The input tunnel appears as a white box with brackets to indicate that it is an auto-indexing tunnel. **Note** A For Loop can process multiple arrays one element at a time using multiple auto-indexing input tunnels. In this situation, the loop uses the smallest array size to determine the number of loop iterations. For example, if two auto-indexed arrays enter the loop with 10 and 20 elements respectively, the loop executes 10 times, processing all elements of the first array but only the first 10 elements of the second array.

③	You can access each individual element of the input array by wiring the auto-indexing tunnel to the code on the subdiagram of the loop. The wire entering the auto-indexing tunnel carries 1D array data, whereas the wire leaving the auto-indexing tunnel carries scalar data. **Note** For multidimensional input arrays, the wire leaving the auto-indexing tunnel carries array data that is one dimension smaller than the input array. For example, if the input array terminal passes a 2D array to the auto-indexing tunnel, the wire leaving the auto-indexing tunnel carries 1D array data.
④	By not wiring a value to the count terminal when auto-indexing is enabled, the loop automatically iterates once for each element in the array. **Note** Wiring a value to the count terminal while auto-indexing is enabled causes the For Loop to use the smallest of the choices between the count terminal and the input array size to determine the number of loop iterations. For example, if an auto-indexed array enters the loop with 10 elements and you wire a value of 15 to the count terminal, the loop executes 10 times.
⑤	If you want to collect the result of each loop iteration in an array, use an auto-indexing output tunnel to pass values out of the loop. To enable auto-indexing for an output tunnel, right-click the tunnel and select **Append Mode** » **Auto Index Values**. If you do not enable auto-indexing, the loop returns only the value from

	the last loop iteration.

Troubleshooting

- If the For Loop iterates an unexpected number of times, note that if you enable auto-indexing for more than one input tunnel or if you wire a value to the count terminal, the actual number of loop iterations becomes the smallest of the choices.
- If the For Loop processes the entire input array in a single loop iteration instead of one element per iteration, verify that the input tunnel is auto-indexing the array. An auto-indexing tunnel appears as a white box with brackets, whereas a non-indexing tunnel appears as a solid box.
- If the execution speed of the loop is too fast, place a Wait node on the subdiagram of the loop to specify a wait period.

Related concepts:

- Loop Timing

Related tasks:

- Repeating Operations
- Repeating Operations a Set Number of Times
- Repeating Operations until a Condition Occurs

Loop Timing

Loop timing refers to how long a loop takes to execute a single iteration. The amount and type of code a loop contains affects its execution speed.

By default, each loop iteration executes as quickly as possible based on the code inside the loop. However, you might want to change the execution speed of a loop for one or more of the following reasons.

Reason to Control Loop Timing	Example Application	Procedure
You want to repeat the code inside a loop and specify a fixed time interval for each iteration.	Take a temperature measurement every 10 minutes.	Adjusting the Execution Speed of a Loop
You want to reduce the execution speed of a loop to make the result of each iteration more easily visible, as seen through indicators on the panel.	Control the rate at which data values are plotted to a chart.	
Instead of allowing a loop to execute at full speed, potentially taking full control of all of the CPU's resources, you want to conserve processing resources for other tasks.	Yield control of the CPU to allow other threads, such as serial or VISA calls, access to processor resources while the loop waits.	
You want to synchronize the execution of multiple loops.	Two loops contain sections of code that take different amounts of time to run. Synchronize these loops to the system clock so that they begin each iteration at the same time.	Synchronizing the Execution of Multiple Loops

Related tasks:

- Adjusting the Execution Speed of a Loop
- Synchronizing the Execution of Multiple Loops

Related information:

- For Loop
- While Loop
- Wait
- Wait Until Next Multiple

Adjusting the Execution Speed of a Loop

By default, each loop iteration executes as quickly as possible based on the code inside the loop. However, you might want to control the rate at which a loop executes in order to specify a fixed time interval for each iteration, reduce the speed at which an indicator changes value, or conserve processing resources.

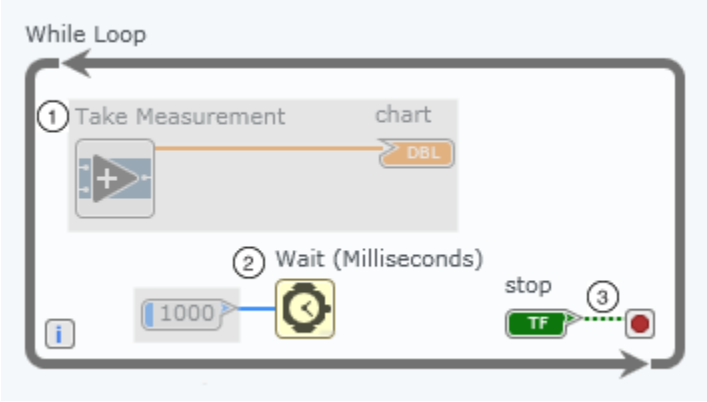
What to Use

- While Loop or For Loop
- Wait

What to Do

Create the following diagram to adjust the execution speed of a loop.

Customize the gray sections for your unique programming goals.



①	Place the code you want to repeat on the subdiagram of a While Loop or For Loop.
②	To specify the amount of time to wait between loop iterations, wire the desired time duration to the input of the Wait node. The Wait node waits until the value of the operating system's

	counter increases by an amount equal to the input you specify. The total running time of the loop is equal to the greater of the time it takes to run the code inside the loop and the wait time you specify.
③	Pressing the stop button does not interrupt the Wait node. For example, if the user presses the stop button in the middle of a loop iteration, the While Loop stops only after the specified wait period has elapsed and the code inside the loop has finished executing.

Troubleshooting

If a loop does not execute at the desired rate, verify that you have specified the input of the Wait node correctly. Consider the following common mistakes:

- If you are converting from another unit of time, verify that your conversion calculation is correct.
- If your code takes longer than expected to execute, note that the total running time of the loop is equal to the greater of the time it takes to run the code inside the loop and the wait time you specify.

Related concepts:

- Loop Timing

Synchronizing the Execution of Multiple Loops

By default, each loop iteration executes as quickly as possible based on the code inside the loop. However, consider a program that includes multiple loops that contain code requiring different amounts of time to execute. You might want to control the rate at which those loops execute in order to synchronize their execution and ensure that the loops begin each iteration at the same time. Use Wait Until Next Multiple to do this.

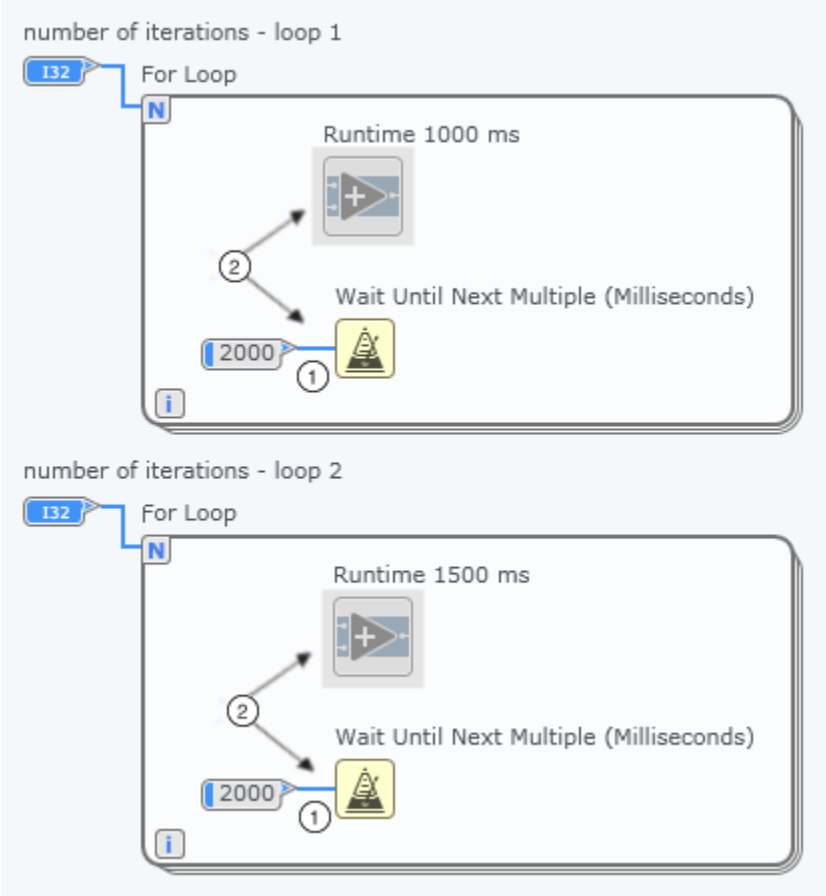
What to Use

- While Loop or For Loop
- Wait Until Next Multiple

What to Do

Create the following diagram to synchronize the execution of multiple loops.

Customize the gray sections for your unique programming goals.



①	Wire the same value to the inputs of the Wait Until Next Multiple nodes placed on the subdiagram of each loop. The loops wait until

	the value of the system clock becomes a multiple of the specified input before beginning each iteration. Therefore, the loops begin each iteration at exactly the same time.
②	When specifying a value for the input of Wait Until Next Multiple, ensure that the value is greater than the time required to execute the code inside the loop. If a loop contains code that takes longer to execute than the time specified, Wait Until Next Multiple has no effect on the execution speed of the loop.

Related concepts:

- Loop Timing

Accessing Data from the Previous Loop Iteration

After a loop completes a single iteration, sometimes you want to use the value of one of its calculations in the next loop iteration. You can use a shift register to pass data from the most recent iteration to the next iteration.

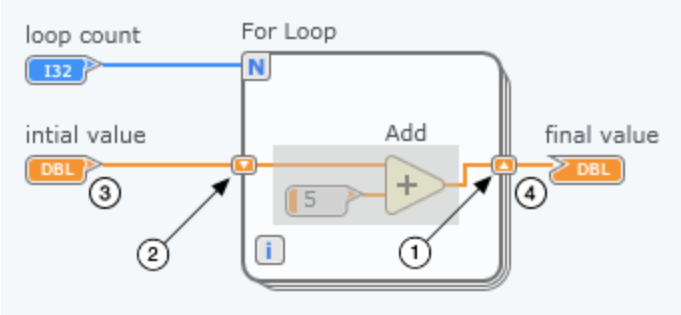
What to Use

- For Loop or While Loop
- Shift register

What to Do

Create the following diagram to pass data from the most recent loop iteration to the next iteration.

Customize the gray sections for your unique programming goals.



①	To share data with the next loop iteration, pass the data into a shift register on the right border of the loop. This right shift register passes that data to a corresponding left shift register. To create a pair of shift registers, right-click the loop border and select **Create Shift Register**.
②	The left shift register contains data passed from the right shift register. Access this data by wiring the output of the left shift register to the code inside the loop.
③	For the first loop iteration, an initialized shift register returns the value wired to its input. If you do not initialize the shift register, it shares the data it held the last time the loop executed, even if that data is from a previous execution of the VI containing the loop.
④	After the loop executes, access the data from the last iteration by wiring the output of the right shift register to the rest of the program. Only values from the last iteration exit the loop through the output of the right shift register.

Examples

In the previous example, the value the left shift register passes into the loop changes after each iteration. The following table records the data each shift register contains after each loop iteration if the **initial value** wired to the left shift register is 10 and the **loop count** is 3.

Shift Register	Value after First Iteration	Value after Second Iteration	Value after Third Iteration
Left shift register	10	15	20
Right shift register	15	20	25

Related tasks:

- Accessing Data from Multiple Past Loop Iterations

Accessing Data from Multiple Past Loop Iterations

A shift register passes values from one loop iteration to the next, but sometimes you need to access values from more than just the previous iteration.

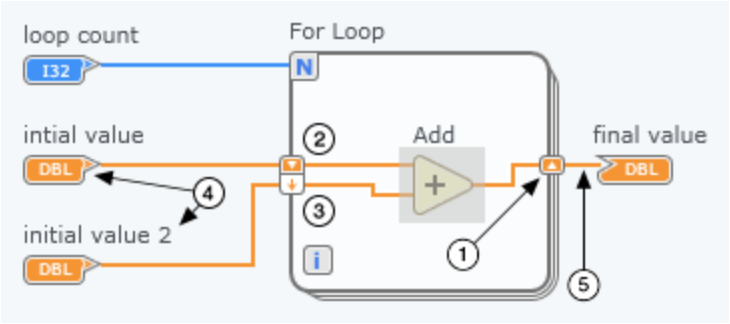
What to Use

- For Loop or While Loop

What to Do

Create the following diagram to pass data from the two most recent loop iterations to the current iteration.

Customize the gray sections for your unique programming goals.



①	A right shift register always passes data to the next loop iteration, regardless of whether you want to share data among one or multiple loop iterations. To create a pair of shift registers, right-click the loop border and select **Create Shift Register**.
②	After each iteration, the first left shift register contains the data passed from the right shift register. Access this data by wiring the output of the first left shift register to the code inside the loop.
③	The second left shift register contains the data from the second most recent iteration.
④	Just like for single shift registers, specify an initial value for each stacked shift register to ensure that each one has a predictable value for the first loop iteration. Each left shift register that lacks an initial value uses the most recent value it contained, even if that value is from a previous loop execution.

⑤	After the loop executes, access the data from the last iteration by wiring the output of the right shift register to the rest of the program. Only values from the last iteration exit the loop through the output of the right shift register.

Examples

In the previous example, the value contained in each shift register changed after each loop iteration. The following table records the data each shift register contains after each loop iteration, assuming the following conditions:

- The **loop count** is 3.
- The **initial value** wired to the first left shift register is 5.
- The **initial value 2** wired to the second left shift register is 2.

Shift Register	Note	First Iteration Value	Second Iteration Value	Third Iteration Value
First left shift register	The first left shift register receives new values from the right shift register.	5	7	12
Second left shift register	The second left shift register receives new values from first left shift register.	2	5	7
Right shift register	The right shift register receives values from the subdiagram of the loop.	7	12	19

Related tasks:

- Accessing Data from the Previous Loop Iteration

Error Management

When an error occurs in your code, the error is logged in the console. You can manage errors programmatically.

Automatic Error Management

If an error occurs while your code is running, the program suspends execution and displays a dialog box with information about the error.

Programmatic Error Management

To collect or process error information while code runs instead of suspending its execution, use error clusters. You can wire error clusters to diagram objects as shown in the following table, but these wiring patterns are not exhaustive or required.

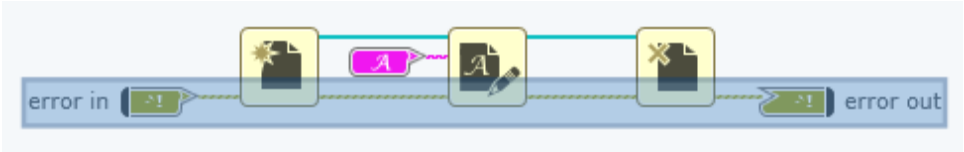
Diagram Object	Description	Example
Nodes	Many nodes include error inputs and outputs to allow for programmatic error handling. Consider wiring these inputs and outputs to implement error handling in your code, especially for I/O operations, such as file I/O, serial, instrumentation, data acquisition, and	

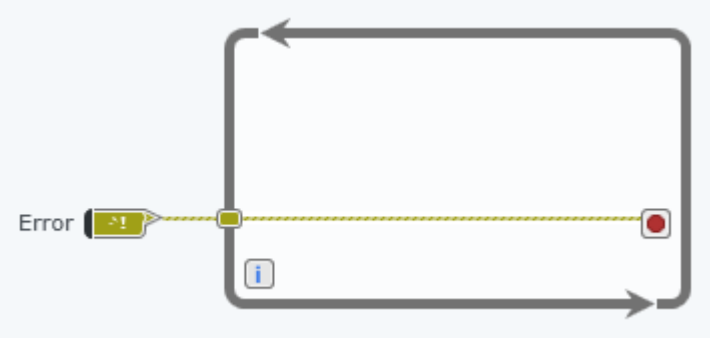
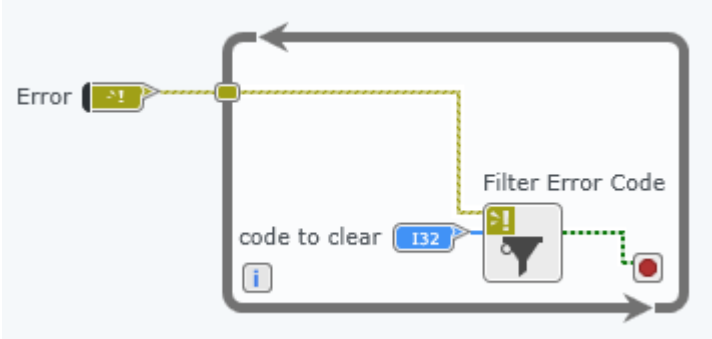
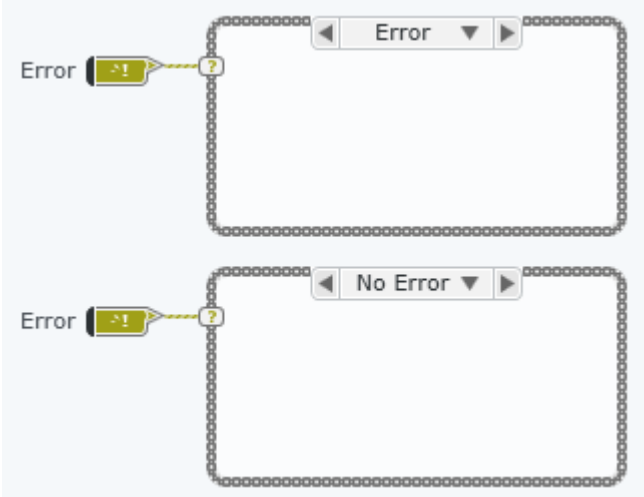
Diagram Object	Description	Example
	communication. As the code runs, each node tests for errors at execution. If no errors occur, the node executes normally. If there are errors, the node that detects the error does not execute and passes the error information to the next node. The next node does the same thing, and so on. At the end of the execution flow, the last node returns error information to the `error out` indicator.	
Loops	The `condition` terminal of a loop can accept an error cluster. To stop the loop when an error occurs, wire an error cluster to the `condition` terminal. You	

Diagram Object	Description	Example
	can also use Filter Error Code to stop the loop when a specific error occurs.	
Case Structures	The `selector` terminal of a Case Structure can accept an error cluster. To execute different sets of code depending on whether an error exists, wire an error cluster to the `selector` terminal. The Case Structure automatically creates two cases, an error case and a no error case.	

Related information:

- Debugging Tools

Executing Code Based on a Condition

Use a Case Structure to incorporate decision logic into your program. A Case Structure contains one or more subdiagrams, or cases, exactly one of which executes when the structure executes.Case Structures behave similarly to `switch` statements or `if-`

`then-else` statements in other programming languages.

What to Use

- Case Structure

> **Tip** If you are choosing between only two values based on a Boolean input, you can use the Select node instead of a Case Structure with a Boolean selector.

What to Do

Create the following diagram to execute different code based on a given condition.

Customize the gray sections for your unique programming goals.

①	Wire the data that you want to use to make a decision to the selector terminal. The data type of the value you wire to the selector terminal defines the various cases that are available in the Case Structure. The selector terminal accepts Boolean, string, integer, floating point, enumerated type, and error cluster data.
②	Use the case selector label to define the various cases in which different code executes. The case name displayed in the case selector label matches the selector terminal value(s) for which the corresponding subdiagram executes. You can define each case using a single value or a range of values. You can also use the case selector label to specify a default case.
③	Place the code you want to execute in each case on the corresponding subdiagram of the Case Structure.

④	To add, duplicate, or delete cases, right-click the border of the Case Structure and select the desired option from the **Cases** shortcut menu.
⑤	To view the different subdiagrams in the Case Structure, click the down arrow on the case selector label and select the desired case.
⑥	Every case must provide data to all output tunnels in order for the Case Structure to execute. If any case does not provide data to an output tunnel, an error occurs, and the output tunnel appears as a white box with a colored border. To resolve this error and allow the Case Structure to execute without explicitly wiring all cases, right-click the unwired output tunnel and select **Default If Unwired**. The output tunnel returns the default value for its data type if the subdiagram that executes does not provide that output tunnel with data. When set to **Default If Unwired**, the output tunnel appears as a white box with a colored border and a dash in the center. When all cases provide data to a particular output tunnel, that output tunnel appears as a solid box.

Troubleshooting

- If the default case executes unexpectedly, verify that the input values wired to the selector terminal match the values in the case selector label exactly.
- An edit-time error occurs when there are values of the selector data type that do not correspond to any subdiagram in the Case Structure. You must either define a default case to handle out-of-range values or create a case for every possible input value. For example, if the selector is an integer data type and you specify cases for 1, 2, and 3, you must specify a default case to execute if the input value is 4 or any other unspecified integer value.

Parsing a String into Smaller Pieces

Parsing a string into smaller pieces allows you to perform operations on individual words or groups of characters in the string. These words or groups of characters are often referred to as ***tokens***. A token is defined as either the next set of characters that

appears before a separator character, called a ***delimiter***, or one of a specified set of operators.

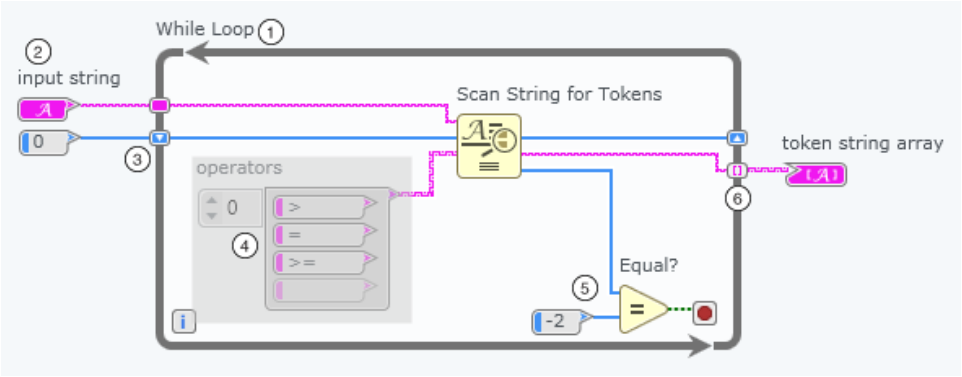
What to Use

- Scan String for Tokens
- While Loop with shift registers

What to Do

Create the following diagram to parse a string into smaller pieces.

Customize the gray sections for your unique programming goals.



①	In order to identify all the tokens in a string, you must use a While Loop. Inside a While Loop, Scan String for Tokens scans the entire input string, returning all tokens it identifies until it reaches the end of the string. If you do not use Scan String for Tokens inside a While Loop, the node stops scanning as soon as it identifies the first token in a string and returns only that token.
②	The `input string` input of Scan String for Tokens contains the string to scan for tokens.

③	To start each scan at the location within the string where the preceding scan ended, pass the `offset past token` output of Scan String for Tokens through a shift register and back into the `offset` input of the same node. Initialize this shift register with the location within the string at which you want to begin parsing. Use 0 if you want Scan String for Tokens to begin its operation at the beginning of the string each time the program runs.
④	Add any strings that you want Scan String for Tokens to identify as tokens to the `operators` input array. The node identifies these strings as tokens even if they are not surrounded by any delimiters.
⑤	Detect the end of the input string by comparing the `token index` output of Scan String for Tokens to -2.
⑥	Use the auto-indexing output tunnel of the While Loop to collect the individual tokens in an array. This array contains all text that appears between delimiters as well as any strings specified in the `operators` input array that are found in the input string. Scan String for Tokens does not return delimiters as tokens but instead uses them to determine where tokens begin and end.

Troubleshooting

- If Scan String for Tokens does not identify a token that you expect it to identify,

check to make sure the `operators` input array does not contain regular expression notation or any invisible characters. Scan String for Tokens does not process regular expressions. Also check for correct capitalization of items in the `operators` input array, as scanning is case-sensitive.

Examples

input string	operators	delimiters	token string	C
4>=0	[>, =, >=]	\s, \t, \r, \n (default)	[4, >=, 0]	If o in s n n o d o S S T c t n t
a==b c!=d	[==, !=]	\s, \t, \r, \n (default)	[a, ==, b, c, !=, d]	
G2 X0.5Y1.0 i0.5j0 z-0.05	[X, Y, Z, i, j, z]	\s, \t, \r, \n (default)	[G2, X, 0.5, Y, 1.0, i, 0.5, j, 0, z, -0.05]	T e a G la c u n

input string	operators	delimiters	token string	C
				c T d a
C1_1.11C2_2.22C3_3.33	None	C, _ (add to `delimiters` array) \s, \t, \r, \n (default)	[1, 1.11, 2, 2.22, 3, 3.33]	T e a fr D w c

State Machine Design Pattern

You can use the state machine design pattern to implement decision making algorithms where a set of distinguishable states exists.

These states, or subdiagrams of code, carry out specific operations within a program. Only one state executes at a time while the machine is active.

After all of the code within a state executes, the state outputs a transition value and initiates a state transition. This transition advances the program from the finished state to the next state indicated by the transition value.

During a state transition, data from the completed state is sent to the upcoming state. Once the state machine executes its final state, data output by the machine becomes available for other parts of the program to use.

When to Use a State Machine

A state machine consists of discrete segments of code, otherwise known as states, that execute one at a time with a transition between each execution.

The following tasks are examples of situations for which state machines are well suited:

- Responding to user interface interactions where the user's action determines which state executes.
- Process testing where each state carries out a step of the process.
- Breaking down difficult to manage applications into smaller, easily maintainable chunks of code.

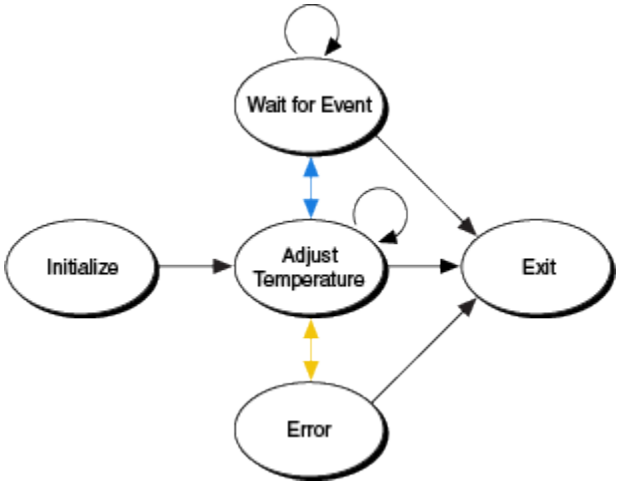
However, a state machine is not well suited for every programming situation. If your program needs to run parallel processes, you may want to choose a different design pattern.

State Diagrams

To keep track of every state and interaction in your program, design a state diagram of your state machine before you start programming.

A state diagram is an illustration of all the states in a state machine and how these states interact with each other. Each circle represents a state, and each arrow represents a possible state transition. While you create your code, you can follow the diagram flowchart to recall how to structure the state machine and how the states within the machine interact.

The following image shows an example of a basic state diagram for a program that changes the temperature of a room over time.



The state diagram shows all possible state transitions in the program. You can use the state diagram to understand how the states within the program interact before analyzing the code that defines the interactions.

- Each arrow points toward the endpoint of a state transition. For example, the initialize state can only transition to the adjust temperature state. On the other hand, the program can transition back and forth between the adjust temperature state and wait for event state.
- A circular arrow, seen on the adjust temperature and wait for event states, indicates that the state can transition to itself.
- A blue arrow indicates that a transition occurs because of a user action, and a yellow arrow indicates that a transition occurs because of an error in the program.

Standard States To Consider When Planning Your Program

When you design a state machine, create a distinct initialize state for the program. You can also add a specific state to handle user input or provide custom error handling, depending on your program needs.

States to Include in Every State Machine You Create

Specify one entry point and one exit point for a state machine to control the code that executes each time the state machine starts up and shuts down.

- **Initialize state**—The first state a state machine executes, which includes any application initialization code. Common uses for initialization code include opening file references and hardware references to use later and bundling control references into the data cluster to unbundle later.

> **Note** Web applications run infinitely once initialized.

A state machine runs continuously until the condition terminal on the While Loop receives the stop value determined by the user. Directing your state machine toward a single exit rather than accounting for multiple exit points allows you to control the shutdown code that executes each time the machine stops. Using a single exit state also helps prevent accidental, premature, or partial state machine shutdowns.

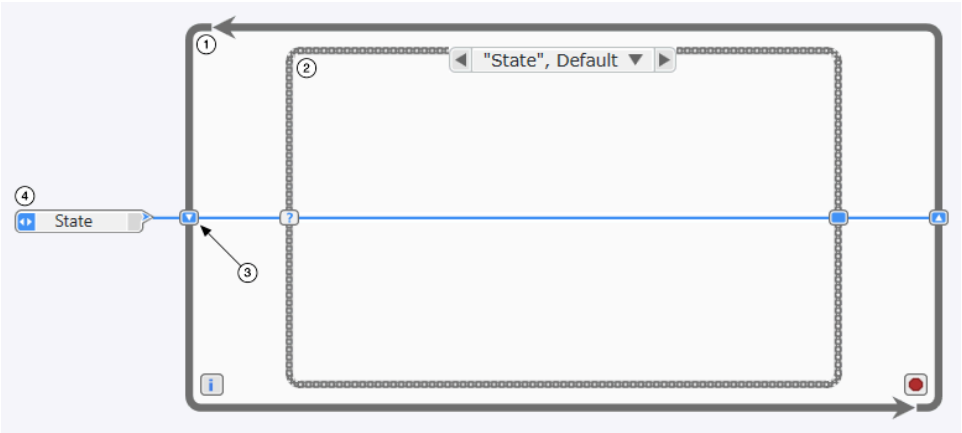
States to Consider Depending on Your Program Needs

If you are designing a program that implements user interface actions or contains specialized error handling, consider including these states in your state machine.

- **Wait for event state**—A state that accepts and implements user input.
- **Error handling state**—A state that contains error handling code for the state machine.

Diagram Components of a State Machine

A state machine has four components on the diagram: a While Loop, a Case Structure, an Enum constant, and a shift register.



1. While Loop—Sets the outer boundary for the state machine code and facilitates state transitions.
2. Case Structure—Contains a subdiagram for each state in the state machine.
3. Shift register—Passes data from a completed state to the upcoming state.
4. Enum constant—Contains a list of every state in the state machine. The state machine uses this constant to update the current state transition value as the program executes.

Note Convert the Enum constant to a G Type and configure its list of values before using it in your state machine. This ensures that every Enum constant in your state machine has a consistent list of values that is easy to update.

Common State Machine Transition Code

Transition code determines which case to execute in the next While Loop iteration. You can use the transitions detailed below as a starting point for the transition code in your program.

Table 1. One State Transition Value

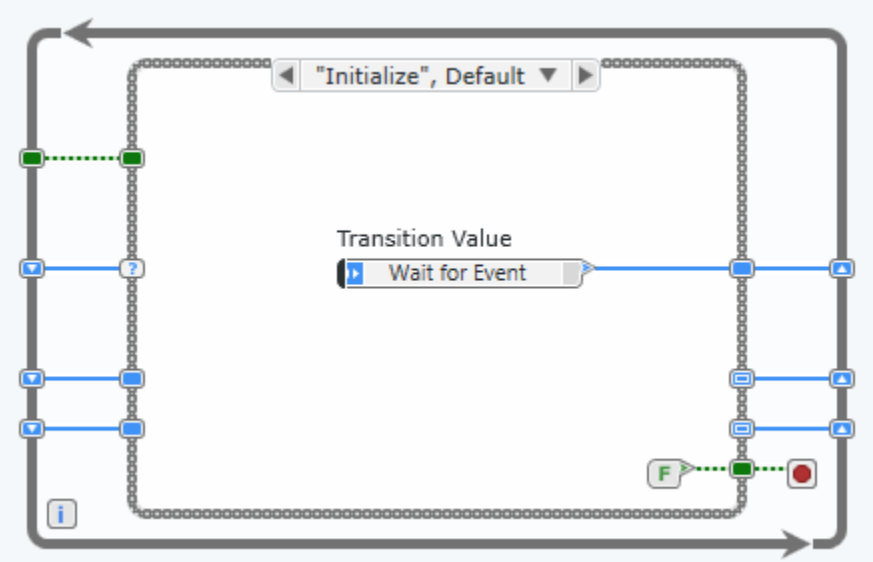
State Transition	Example Image
Use an Enum constant transition when you always want to transition to a specific state after the code in the current state executes. For example, the Initialize and Exit states in a state machine often use an Enum constant transition.	

Table 2. Two State Transition Values

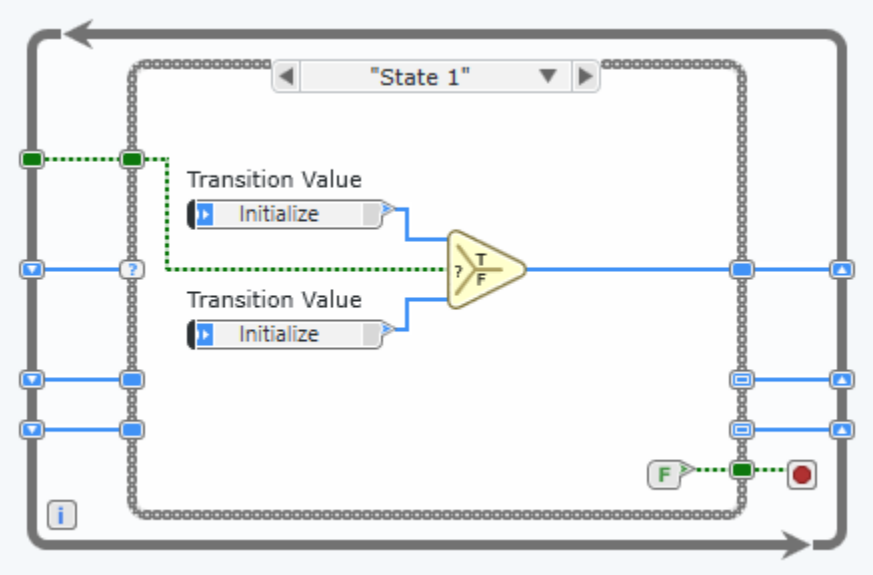
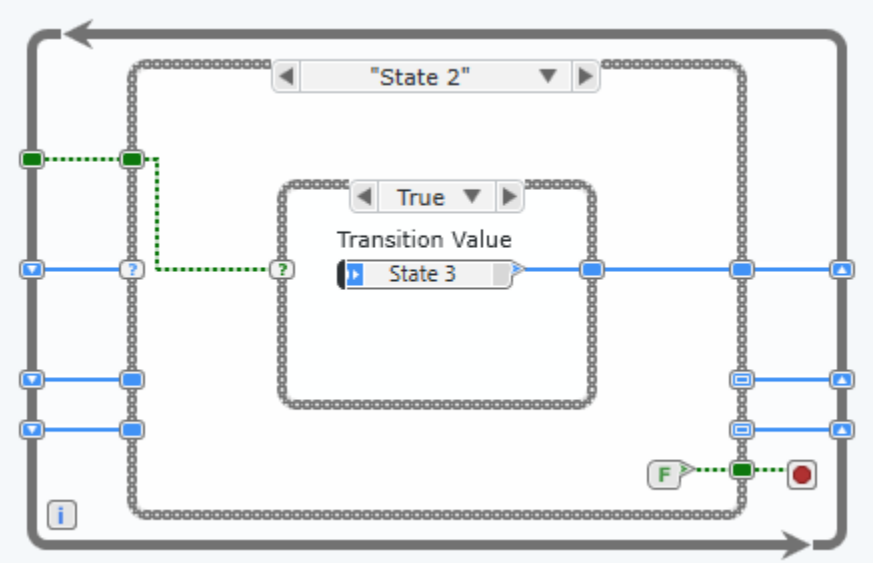
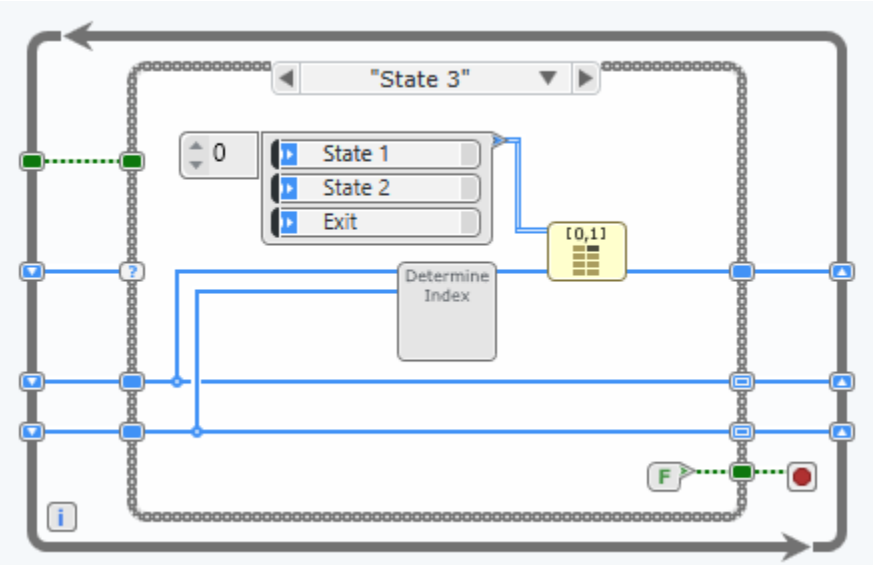
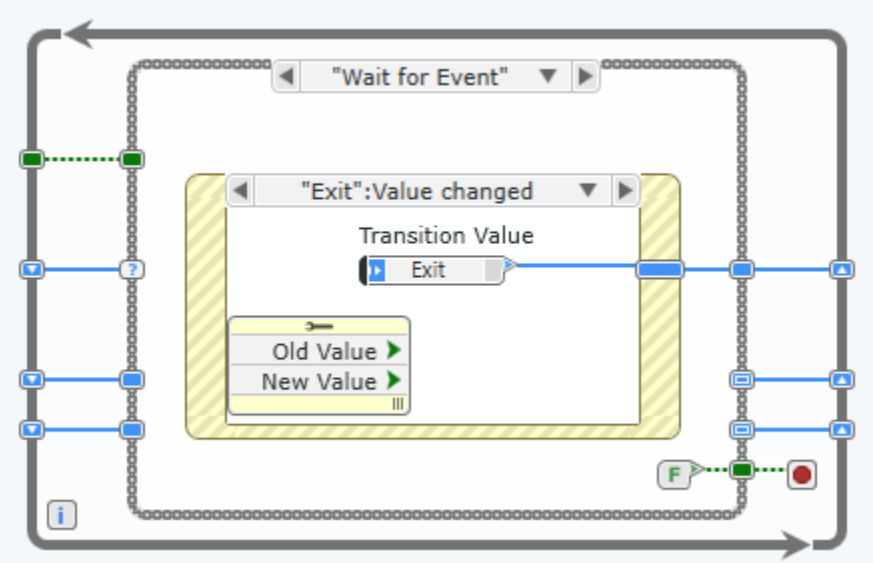
State Transition	Example Image
Use a Select node and a Boolean value to choose between one of two transition values.	

Table 3. Three or More State Transition Values

State Transition	Example Image
Use a Case Structure transition when you need to handle a variety of situations. Add a case for each transition you need to program and create transition code in each case to adjust the transition value when the case executes. Each case in the Case Structure can include a single transition value or multiple values that the program must choose between.	
Use a transition array when you need to choose between more than two transition values at one time. At run time, an Index Array node selects a transition value from an array constant containing each possible state transition value.	

State Transition	Example Image
Use an Event Structure when you need to change states in response to a user action in your program. The Event Structure monitors the program for events and executes a subdiagram of code when a specific event occurs. Each event in the Event Structure can include a single transition value or multiple values that the program must choose between. For example, the Wait for Event state in the simple state machine template uses an Event Structure. **Note** If you need to perform an action periodically in the Wait for Event state, such as updating a timer, you can include an application timeout event in the Event Structure.	

Best Practices for Creating Projects in G Web Development Software

To develop projects that successfully serve your needs and the needs of your end users, refer to the following best practice guidelines:

- File and Project Organization for Projects in G Web Development Software

- Icons and Connector Panes for Projects in G Web Development Software
- Panel Design for Projects in G Web Development Software
- Diagram Design for Projects in G Web Development Software
- Other Best Practices for Projects in G Web Development Software

Related reference:

- File and Project Organization in G Web Development Software
- Icons and Connector Panes for G Web Development Software Projects
- Panel Design for G Web Development Software Projects
- Diagram Design for G Web Development Software Projects
- Localization for LabVIEW NXG Projects
- Other Best Practices for LabVIEW NXG Projects

File and Project Organization in G Web Development Software

Refer to the following table for best practices for organizing files and projects in G Web Development Software.

Guideline	Required or Recommended?	Details	Example(s)
Avoid using special characters in file names.	Recommended	Using special characters in file names can cause compatibility concerns across platforms.	Avoid the following: • File separators such as colons, forward slashes, and backward slashes • Non-alphabetical and non-numerical symbols, such as the trademark symbol • Punctuation marks, such as parentheses, quotation marks, brackets, and operators • White space characters, such as tabs and new lines

Guideline	Required or Recommended?	Details	Example(s)
			Note You can use spaces for most applications, but avoid using spaces in a top-level `.gviweb` to create a human-readable URL.
Make sure your project organization is logical and easy to use.	Recommended	• Create a hierarchical structure with easily accessible top-level VIs. • Place support VIs in folders within the project and group them to reflect modular components, such as instrument drivers, other drivers, and configuration utilities. • Limit the number and the levels of directories you use in a project.	N/A

Icons and Connector Panes for G Web Development Software Projects

Refer to the following table for best practices for creating icons and connector panes in G Web Development Software.

Guideline	Required or Recommended?	Details	Example(s)
Consider using the well-designed icons in the G Web Development Software libraries as prototypes for your icon.	Recommended	If you cannot find or create a picture to use for an icon, you can use text.	N/A
Create a uniform icon style for all related VIs.	Recommended	A uniform style helps users visually determine which subVIs are associated with a top-level VI.	N/A
Avoid using colloquialisms when making icons.	Recommended	Colloquialisms, even in pictures, are difficult to translate. Users that speak languages other than English may not understand a picture that relies on cultural background knowledge.	If you represent a data logging VI with a picture of a lumberjack, some users may not know the cultural reference to understand how a lumberjack can represent data logging.
Create a meaningful icon for every VI.	Recommended	The icon represents the VI on a palette and diagram. Well-designed icons help users gain a better understanding of the subVI without the need for excess documentation.	N/A

Guideline	Required or Recommended?	Details	Example(s)
Consider common node sizes in LabVIEW NXG when designing an icon.	Recommended	Refer to the following common node sizes: • 30x30 for small nodes. This size supports the 3-1-1-3 connector pane pattern. • 40x40 for the default VI size. This size supports the 4-2-2-4 connector pane pattern. • 50x50 for the default size used by NI hardware driver and Advanced Analysis Library APIs. This size supports the 5-3-3-5 connector pane pattern.	N/A
Make sure to set inputs and outputs for each subVI as required, recommended, or optional in the connector pane for that subVI, and make sure the settings make sense for your project.	Recommended	Use the **Usage** settings on the **Item** tab for the connector panes of all subVIs to specify the input and output settings for each subVI. The **Usage** setting for connector pane terminals affects the appearance of the inputs and outputs in the Context Help and reminds users to wire subVI connections.	N/A

Guideline	Required or Recommended?	Details	Example(s)
		Use the required setting for inputs that users must wire for the subVI to run properly. Use the optional setting for inputs that have default values that are appropriate for the subVI the majority of the time. Use the recommended setting for all other inputs.	
Assign inputs and outputs according to the way a user will eventually wire VIs together.	Recommended	Wire inputs on the left and outputs on the right of the connector pane because standard data flow moves from the left to the right. Consistency between the location you assign inputs and outputs in the connector pane and their location in the actual data flow promotes ease of use and reuse.	• If you create a group of subVIs that you often use together, give the subVIs a consistent connector pane with common inputs in the same location to help you remember where to locate each input. • If you create a subVI that produces an output that another subVI uses as an input, such as references, task IDs, and error clusters, align the input and output connections to simplify the wiring patterns. • Assign two inputs of a VI to the left two terminals of the corresponding connector pane and two outputs of that VI to the right two terminals of the corresponding connector pane.

Guideline	Required or Recommended?	Details	Example(s)
Reserve the bottom left and right connector pane terminals for the error input and error output.	Recommended	Session-based APIs are the only exception to this recommendation. For session-based APIs, you can place error inputs and outputs directly beneath session and reference inputs and outputs.	N/A
Avoid creating connector panes with more than 16 terminals.	Recommended	Including too many terminals can make a subVI difficult to understand. You can consider either splitting the functionality of the subVI into multiple subVIs or using clusters to create logical groupings of the inputs to the subVI, depending on which solution makes sense for your project.	N/A
If your subVI includes a pass-through input and output pair, add an in suffix to the control and an out suffix to the indicator.	Recommended	Adding these suffixes to pairs of inputs and outputs indicates a relationship between the inputs and outputs.	If you name an input reference in, name the related output reference out.
If your subVI includes a pass-	Recommended	You can exclude references from this	N/A

Guideline	Required or Recommended?	Details	Example(s)
through input and output pair, and you wire the control directly to the indicator on the diagram to prevent the value from changing, remove the indicator from the connector pane.		guideline.	

Related information:

- Configuring an Existing VI for Use As a SubVI
- Creating a SubVI

Panel Design for G Web Development Software Projects

Refer to the following table for best practices for designing panels in LabVIEW NXG

Guideline	Required or Recommended?	Details	Example(s)
Avoid using all capital letters in labels or panel documentation.	Recommended	Capital letters can make text seem more important than necessary.	N/A
Position the panel in the top left, spatially even with the controls palette.	Recommended	If you position the panel in the top left, you can prevent the user from opening the	N/A

Guideline	Required or Recommended?	Details	Example(s)
		panel to a position that is potentially off the screen or otherwise difficult to see and read.	
Display the labels of all controls and indicators on the panel.	Recommended	Make sure the labels of all controls are meaningful to increase clarity and ease of use for users.	N/A
Set reasonable default values for controls. **Note** Default values you set for controls automatically append to the terminal name. The Context Help displays the default value when you hover over the control.	Recommended	Make sure the default values you set do not generate errors when you run the top-level VI. **Note** When possible, avoid setting default values for indicators such as graphs, arrays, and strings. Setting default values for those types of indicators	N/A

Guideline	Required or Recommended?	Details	Example(s)
		wastes disk space when you save the VI.	
Use default values strategically and logically.	Recommended	Planning the use of default values can save space and simplify code.	N/A
Avoid displaying labels on the panel for buttons that display Boolean text.	Recommended	Only display the Boolean text for these buttons. **Note** If you click the boolean text of a checkbox, the value of that checkbox toggles. The value of that checkbox does not toggle if you click the label.	N/A
Format any text on your panel appropriately.	Recommended	• Use default fonts when possible. • When the	N/A

Guideline	Required or Recommended?	Details	Example(s)
		alignment of characters is critical, use monospace fonts for string controls and indicators, and space the characters equally. • For free labels, only use carriage returns to separate paragraphs. • Resize labels to enable word wrapping. • Include extra space in free labels to allow for longer or larger strings due to font differences in localized languages.	
Group and arrange controls logically and aesthetically.	Recommended	• Consider the arrangement of controls on panels, and keep panels simple to avoid confusing users. • For top-level VIs that users can see, place the most important controls in the most	N/A

Guideline	Required or Recommended?	Details	Example(s)
		prominent positions. • For subVI panels, place controls and indicators of the subVI to correspond with the connector pane pattern. **Note** Regardless of the location of the error cluster connector pane connection, place error cluster controls and indicators at the bottom of the panel.	
Use the **Align** and **Distribute** options in the **Layout** pull-down menu to create a uniform layout.	Recommended	N/A	N/A
Visually group objects	Recommended	• Use decorations	N/A

Guideline	Required or Recommended?	Details	Example(s)
with related functions.		from the Drawings palette. • Use clusters to group related data. Avoid using clusters for only aesthetic purposes.	
Configure path inputs and outputs appropriately.	Recommended	• Use path controls instead of string controls to specify the location of files or directories. Path controls and indicators work similarly to strings, but the software formats paths using the standard syntax for the platform you are using. • Avoid hiding the **Browse** button on path controls. • Set the browse action correctly for the **Browse** button on path controls in the **Item** tab.	If you set a browse action in which a user needs to select a directory, select **Select Folder** from the **Action** drop-down menu in the **Item** tab.
Determine whether an enum or ring is more effective than a Boolean.	Recommended	An enum may be more efficient in cases where two states may increase to more	N/A

Guideline	Required or Recommended?	Details	Example(s)
		states.	
Arrange items in a cluster vertically.	Recommended	To arrange items vertically, select **Vertical** from the **Arrange** drop-down menu in the **Item** tab.	N/A
Use imported graphics to enhance the panel. **Note** This guideline is specific to user interfaces.	Recommended	Import graphics and text objects to use as panel backgrounds by dropping URL Image from the Decorations palette and navigating to the image on the **Item** tab.	N/A
Use color logically, sparingly, and consistently, if at all. **Note** This guideline is specific to user interfaces.	Recommended	• Never use color as the sole indicator of device state. • Use a minimal number of colors, emphasizing black, white, and gray. Color can be difficult to discern or distract the user from important information. • Use light gray, white, or pastel colors for backgrounds. Use bright,	• A yellow, green, or bright orange background makes it difficult to see a red danger light. • People with some degree of color-blindness can struggle to detect certain color changes. You can upload an image of your panel to an online color blindness simulator to test your panel under various degrees of color blindness.

Guideline	Required or Recommended?	Details	Example(s)
		highlighting colors only when a term is important, such as an error notification. • Because multi-plot graphs and charts can lose meaning when displayed in black and white, use different plot styles, such as dots, and dashes, in addition to color to further differentiate multiple plots.	

Diagram Design for G Web Development Software Projects

Refer to the following table for best practices for designing diagrams in G Web Development Software.

Guideline	Required or Recommended?	Details	Example(s)
Use a type definition when you use the same unique control in more than one location or when your project includes a large data structure that passes between	Required	A type definition automatically propagates changes to the control or data structure throughout all relevant VIs and subVIs.	N/A

Guideline	Required or Recommended?	Details	Example(s)
several subVIs.			
Select **Autosize list view** in the **Item** tab for any properties nodes, such as Cluster Properties or Waveform Properties.	Required	The **Autosize list view** option ensures that all element names are fully visible.	N/A
Make sure that data flows from left to right and that wires enter the diagram from the left and exit to the right.	Recommended	Although the positions of program elements do not determine execution order, avoiding right-to-left wiring helps users read and comprehend the diagram. **Note** Only wires and structures determine execution order.	N/A
Avoid creating any backwards wires.	Recommended	Wires that cause data to flow from right to left contradict style best practices.	N/A
For subVIs and top-level VIs that do not contain loops, place	Recommended	Placing controls and indicators in these positions increases	N/A

Guideline	Required or Recommended?	Details	Example(s)
controls on the far left and indicators on the far right.		readability and usability by giving users a familiar and expected location to look for the controls and indicators.	
For non-user interface subVIs, make sure all controls and indicators that are on the connector pane also reside on the top-level diagram.	Recommended	N/A	N/A
When you use enumerated type constants, always create G Types of those constants.	Recommended	• Enumerated type constants are useful for making diagram code easier to read. When you wire an enumerated type constant to a Case Structure, the string labels appear in the selector label of the Case Structure. • G Types prevent you from needing to rewrite code each time you add or remove an item from an enumerated type constant.	N/A

Guideline	Required or Recommended?	Details	Example(s)
Use a docked constant if the only purpose of the constant is to define a data type and the value is unimportant.	Recommended	Docked constants that contain meaningful values can inhibit the readability of the diagram. If a docked constant is set to the default value, it will appear hollow. If any other value is used, it will be a solid color.	N/A
If you display the list view of a node, make sure the entire name of the longest item in the node is visible.	Recommended	Also, resize the node for shorter names to reduce any extra space.	N/A
Use list view for all nodes imported with the Jenkins Shared Library Interface (JSLI) document type, and show the node label.	Recommended	List view helps users distinguish between JSLI nodes and subVI nodes and also improves readability.	N/A
If you wire an enum to a Case Structure, make sure none of the cases are marked as the default case.	Recommended	One exception to this guideline is if the Case Structure only operates on one or a small number of the overall number of items in the enum.	N/A
If you wire a ring,	Recommended	Removing additional	See the ***Case Values of a***

Guideline	Required or Recommended?	Details	Example(s)
string, or numeric to a Case Structure, remove any additional values from the string in the Case Selector Label.		values increases clarity.	***Case Structure*** section following this table for a visual example.
If possible, avoid using ellipsis notation for enum values.	Recommended	Enumerate all the values so that if you add a new value later, the diagram breaks and you can consciously handle the new enum value. Use range notation sparingly.	N/A
If your VI includes its own clusters or enums on the diagram, define them with type definitions.	Recommended	Type definitions allow you to manage these data types in one place and ensure that any changes you make to a data type propagate across all relevant VIs and subVIs.	N/A
Use comments to document the functionality of your code.	Recommended	N/A	N/A
Reference applicable VIs in comments instead of showing the labels on all subVIs in a	Recommended	When you display object labels, you cannot reposition them, so your diagram	N/A

Guideline	Required or Recommended?	Details	Example(s)
diagram.		width must increase unnecessarily.	
Consider whether or not your VI needs an error Case Structure surrounding its contents.	Recommended	If the error Case Structure is not handling critical running code, consider simply passing the error wire through all diagram nodes without creating a case around the whole diagram. You may want to use an error case if your VI is manipulating the error cluster at all. Using an error case allows you to case out the relevant diagram so that your error manipulation code does not modify any error entering the VI.	High iteration loops and modal dialogs are examples of critical running code.
Arrange diagrams so that the primary wire travels in a straight line between nodes.	Recommended	The primary wire is typically a reference or error wire.	N/A
Keep approximately 20 pixels of white space between objects.	Recommended	An overcrowded diagram without adequate white space is difficult to read.	N/A

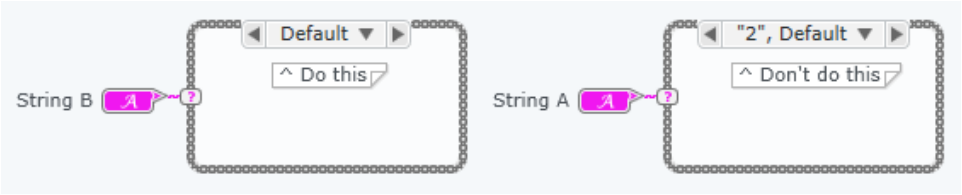
Guideline	Required or Recommended?	Details	Example(s)
Expand array constants to display all array elements, plus one empty element at the end.	Recommended	Displaying all array elements with an empty space at the end helps users see the entirety of the array. If you do not have enough space for a large array constant, display the scrollbar.	N/A
Use a comment to label long wires and identify their use.	Recommended	Wire comments are useful for identifying and describing wires that come from shift registers and long wires that span the entire diagram.	N/A
Avoid bending wires when possible, and keep wires short.	Recommended	Wires with long, complicated paths are difficult to follow.  **Note** Avoid replacing long wires with local or global variables.	N/A
Align and distribute nodes, terminals, and constants.	Recommended	When you align and distribute objects evenly, you can use straight wires to connect objects and	N/A

Guideline	Required or Recommended?	Details	Example(s)
		create readable diagrams.	
Avoid placing diagram objects, such as subVIs or structures, on top of wires, and avoid placing any wires under diagram objects because the software can hide certain segments of the resulting wire.	Recommended	Draw wires so that you can clearly see if a wire connects to a terminal. Avoid wiring through structures if the structure does not use the data in a wire.	N/A
Make sure control and indicator terminals on the connector pane do not reside inside structures on the diagram.	Recommended	The development environment can optimize subVI execution time when all controls and indicators on the connector pane reside on the top-level diagram of the VI.	N/A
Save the VI with the most important frame of multi-framed structures, such as a Case Structure, displayed.	Recommended	Saving the VI with the most important case displayed increases readability because the user does not have to switch cases immediately.	If you open a diagram and the error case that wraps the entire diagram is displayed first, you must switch to the no error case to see the actual diagram code.
Maximize the performance of code	Recommended	When a program has large arrays or critical	Array data types can affect the memory usage of an

Guideline	Required or Recommended?	Details	Example(s)
on the diagram.		timing problems, you can use the following guidelines to maximize the performance of the VI: • If possible, avoid building arrays using Build Array within a loop because the node makes repetitive calls to the memory manager. Instead, use auto-indexing or pre-size the array and use Replace Array Subset to place values in the array. Also avoid using Concatenate Strings with strings because, in memory, the software handles strings as arrays of characters. • Choose the proper array data type to control the memory usage of the application. • Display only necessary information on the panel. Send data to indicators only if	application. For example, if your double-precision, floating-point array of 100,000 values is actually storing single-precision, floating-point values, you are using memory inefficiently. In this case, use an array of single-precision, floating-point values to match the data type stored in the array and reduce the memory usage.

Guideline	Required or Recommended?	Details	Example(s)
		the data is different from what the indicator already displays. Frequently updating panel indicators with new data can affect the performance of the VI, especially if you display large amounts of data in graphs or charts.	
If your VI includes several I/O name controls, such as Waveform and Digital Waveform, that are stacked vertically, configure them to not show type.	Recommended	If you set I/O name controls to not show their type, you can maximize information density and avoid overlapping the controls.	N/A

Case Values of a Case Structure



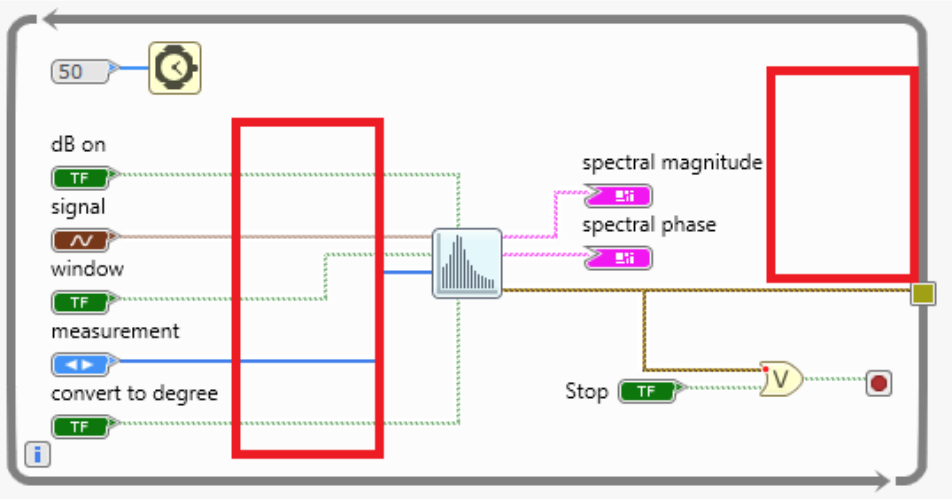
Localization for LabVIEW NXG Projects

Refer to the following table for best practices for localization in LabVIEW NXG.

Guideline	Required or Recommended?	Details	Example(s)
Position plot legends to avoid any potential overlaps.	Required	Plot legends expand to the right when plot names grow due to larger system fonts or longer localized text.	N/A
On the panel, check for consistent placement of control labels, and allow for extra space between controls to prevent labels from overlapping objects due to localization concerns.	Required	N/A	• If you place a label on the left of an object, make sure to justify the label to the right and create some space to the left of the text. • If you center a label over or under an object, make sure to center the text of that label as well.
On the diagram, create extra space inside free labels to account for longer or larger strings due to font differences and localization.	Required	N/A	N/A
If you localize a project or library that contains icons with text, make sure you also localize the text on the icon.	Required	N/A	N/A

Guideline	Required or Recommended?	Details	Example(s)
Create extra space in areas of a panel or diagram where text may grow due to larger system fonts or localized text.	Required	N/A	You can leave extra white space for control and indicator labels to potentially grow in size. See the ***Spacing for Panels and Diagrams*** section following this table for a visual example.
Avoid using enums in shipping content if possible.	Required	Although enum labels on the panel and diagram are localized, individual enum items are not.	N/A

Spacing for Panels and Diagrams



Other Best Practices for LabVIEW NXG Projects

Refer to the following table for LabVIEW NXG best practices.

Guideline	Required or Recommended?	Details	Example(s)
Avoid using absolute paths in VIs.	Required	Absolute paths may cause problems when you build an application or run the VI on a different computer. If you must use an absolute path, make sure that you include code to test that the path exists and to create the path if it does not exist.	N/A

Best Practices for Designing and Developing an Application Programming Interface (API) in G Web Development Software

To develop an API to distribute to other users that is consistent with NI style recommendations for G content, refer to the following best practice guidelines:

- File Organization and Node Naming
- Component Organization
- Icons and Connector Panes
- Panel Design
- Data Type Selection
- Palette Taxonomy
- Documentation
- Error Message Design
- API Design

You also need to follow the Best Practices for Designing and Developing Projects in G Web Development Software if you want your API to be consistent with NI best practice recommendations for G content.

Throughout this section, the term ***required*** refers to a guideline that NI requires when designing an API. The term ***recommended*** refers to a guideline that NI suggests when designing an API.

File Organization and Node Naming for Distributed APIs

Refer to the following table for best practices for organizing files and naming nodes.

Guideline	Required or Recommended?	Details	Example(s)
Use title case.	Required	Capitalize the following in subVI names: • The first and the last word • All nouns, pronouns, adjectives, verbs, adverbs, and subordinate conjunctions (***as***, ***because***, ***although***)	***General Error Handler***, not ***General error handler***
Capitalize the second part of a hyphenated compound.	Required	N/A	***Fixed-Point***, not ***Fixed-point***
Do not capitalize articles (***a***, ***an***, ***the***), coordinate conjunctions (***and***, ***or***, ***nor***), or prepositions regardless of length, unless they are the first	Required	Boolean operators are an exception. Write Boolean operators, such as ***AND*** and ***OR***, in all capital letters.	***Search and Replace***, not ***Search And Replace***

Guideline	Required or Recommended?	Details	Example(s)
or last word.			
Do not capitalize ***to*** in an infinitive phrase unless ***to*** is the first word node name.	Required	N/A	***Path to String***, not ***Path To String***; ***To String***, not ***to String***.
Use industry-standard capitalization for an engineering or scientific term.	Required	N/A	***PolyBezier***, not ***Polybezier***
Write acronyms in all capital letters.	Required	N/A	***FIFO***, not ***Fifo***, ***URL***, not ***Url***
Use spaces between words.	Required	N/A	***Feedback Node***, not ***FeedbackNode***
Do not use special characters.	Required	Boolean operators are an exception. Use a question mark (?) for node names when the primary output is Boolean, such as ***Equal?***, not ***Is Equal***.	***Search and Replace***, not ***Search & Replace***
Make sure none of the nodes or VIs in your API share the same name with a node or VI that	Required	Unique node and VI names in an API make unique palette object names for Quick Drop users.	N/A

Guideline	Required or Recommended?	Details	Example(s)
is already in the LabVIEW NXG palettes.			
Do not use the forbidden word, ***Example***, in the names of your API nodes or VIs.	Required	N/A	N/A

Related concepts:

- Best Practices for Designing and Developing an Application Programming Interface (API) in G Web Development Software

Related reference:

- File and Project Organization in G Web Development Software

Component Organization for Distributed APIs

Refer to the following table for best practices for organizing components.

Guideline	Required or Recommended?	Details
Set the `Exported` or `Non-exported` designation of the VIs in your API appropriately.	Required	The `Exported` designation indicates to users which VIs are guaranteed to have a consistent interface.
Mark a VI as `Exported` only if it is a public member of your API.	Required	Mark a VI as `Exported`.

Guideline	Required or Recommended?	Details
If a G Type is on the connector pane of an exported VI in your API, also mark that G Type `Exported`.	Required	N/A
Follow the namespacing guideline `[Company].[Product].[Component].gcomp` for your API component.	Required	For example, `NI.G Core.Data Type.gcomp` is a component NI owns, part of the G Core product (in other words, the core libraries that compose the G language), and these VIs pertain to parsing Data Types.

Related concepts:

- Best Practices for Designing and Developing an Application Programming Interface (API) in G Web Development Software

Icons and Connector Panes for Distributed APIs

Refer to the following table for best practices for creating icons and connector panes.

Table 4. Guidelines for SubVI Size

Guideline	Required or Recommended?	Details
Make all VIs or as many VIs as possible the same size.	Required	The icon size is determined by the connector pane pattern that the subVI requires. Refer to the icon and connector pane guidelines in Icons and Connector Panes for G Web Development Software Projects for more information.
When resizing a VI, expand it in	Required	The direction a VI can stretch depends on error input and output

Guideline	Required or Recommended?	Details
one direction.		location on the connector pane. • If the error input and output are beneath the instrument handles, as with driver APIs, vertically expand the VI. • If the error input and output are on the lower-left or lower-right of the node, horizontally expand the VI. Increase the size of the entire API to be more consistent, if necessary.

Table 5. Guidelines for Iconography

Guideline	Required or Recommended?	Details	Example(s)
Make the node size big enough to communicate its functionality through an icon.	Required	The icon should communicate node behavior.	N/A
Make consistent iconography for all the VIs in your API.	Required	Default glyphs are in the G Web Development Software project on the Icon tab. You can use the default glyphs as a starting point in your VIs.	N/A

Table 6. Guidelines for Interface Elements

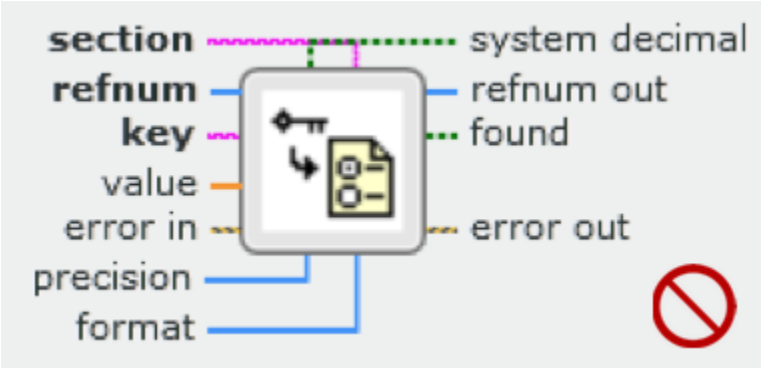
Guideline	Required or Recommended?	Details	Example(s)
Include pass-through wires in the upper corners of a VI only if it is part of a reference-based API.	Required	A reference value is not modified during execution, so passing it through VIs in an API is convenient. A VI could potentially modify a data value, in other words, any wire that is not a reference or a cluster/class of references, if it is passing through. Pass data values through VIs only if the VI might modify the value of the wire.	Reference-based APIs in G Web Development Software include queue and HTTP.
Either coerce numeric inputs into a range that your code expects, or handle out-of-range values with an error condition or warning.	Required	Do not configure data limits for numeric controls.	If you specify the speed of a motor, and the motor can go only 0-100 mph, make sure you are either coercing the input value with an In Range and Coerce function to be within 0-100, or return an error if you specify a value outside that range.
Use lower case for input and output names.	Required	N/A	Use: ***exported waveform*** Do not use: ***Exported Waveform***
Use input and output names that are easy to understand.	Required	N/A	Use: ***remainder*** Do not use: ***x-y*floor(x/y)***

Guideline	Required or Recommended?	Details	Example(s)
Name inputs and outputs consistently.	Required	N/A	For three inputs that have the same data type and source type, use: ***task in, task in, task in***. Do not use: ***task in, myDAQ task in, myDAQ resource in***
Use simple words, not symbols.	Required	N/A	Use: ***number of samples*** Do not use: ***# of samples***
Use a question mark (?) for input and output names when the data type is Boolean with an implied question.	Required	N/A	Use: ***UTC?*** Do not use: ***is UTC***
Do not use a question mark (?) for Boolean inputs and outputs that are commands.	Required	N/A	Use: ***reset*** Do not use: ***reset?***
Do not include default values in parentheses.	Required	Set default values in the configuration panel. G Web Development Software automatically appends the default values to the label.	Use: ***max characters per row*** Do not use: ***max characters/row (no limit:0)***

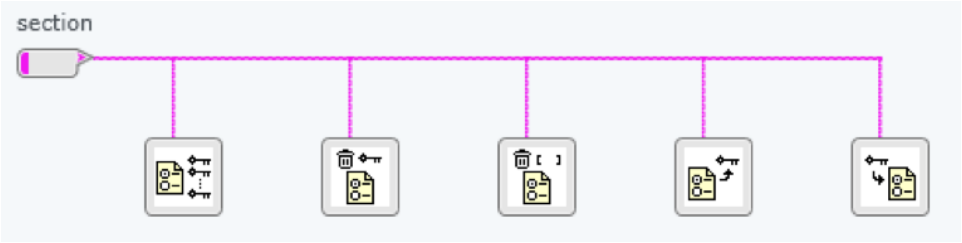
Guideline	Required or Recommended?	Details	Example(s)
Do not include units of measure in parentheses.	Required	Set units in the configuration panel. G Web Development Software automatically appends the units to the label.	Use: ***delay time*** Do not use: ***delay time (s)***
Make sure the names of inputs and outputs of your VIs use complete English words, unless users worldwide know an abbreviation.	Required	N/A	N/A
If you use the top or bottom inputs and outputs on the connector pane, make sure their wires never cross each other.	Required	N/A	See the ***Avoid Crossing Wires*** section at the end of this table for a visual example.
Assign the input and output terminals on the connector pane in a way that simplifies the wiring diagram.	Required	Consider how users might wire the VIs together when you assign terminals. Align the output terminals with the corresponding input terminals.	N/A
If multiple VIs in your API have the same input or output, place	Required	N/A	See the ***Locate Inputs and Outputs Consistently*** section at the end of this

Guideline	Required or Recommended?	Details	Example(s)
all similar inputs and outputs in the same location on the connector pane of all VIs.			table for a visual example.
Do not leave any terminals of your public API VIs in the unplaced items tray.	Recommended	When all controls/ indicators are on the panel, the user can open the panel of your VI and see input and output values during debugging.	N/A
Set the display format of numeric controls and indicators to produce reasonable format settings for controls and indicators that users create from them.	Recommended	The display format settings of a source numeric control or indicator are passed on to any controls and indicators created from them.	N/A

Avoid Crossing Wires



Locate Inputs and Outputs Consistently



Related concepts:

- Best Practices for Designing and Developing an Application Programming Interface (API) in G Web Development Software

Related reference:

- Icons and Connector Panes for G Web Development Software Projects

Panel Design for Distributed APIs

Refer to the following table for best practices for designing panels.

Guideline	Required or Recommended?	Details
Use the flat style without shadows for controls and indicators on the panels of your public API VIs.	Required	Create a consistent style for users when they create controls and indicators from the inputs and outputs of your VIs.

Related concepts:

- Best Practices for Designing and Developing an Application Programming Interface (API) in G Web Development Software

Related reference:

- Panel Design for G Web Development Software Projects

Data Type Selection for Distributed APIs

Refer to the following table for best practices for selecting data types.

Guideline	Required or Recommended?	Details	Example(s)
For numeric input or output values, use double-precision for floating-point values and I32 for integer values unless there is a specific reason to use another type.	Required	N/A	N/A
Always use an error cluster for reporting error conditions. Never use an error code or error Boolean output by itself.	Required	G Web Development Software has a single Error API for error handling. If API VIs do not use the error cluster to convey error information, you cannot use the Error API for handling errors.	N/A
Use a text ring only for parameters that have a natural association to several discrete possibilities and no numeric content.	Recommended	An exception for numeric content is if the set of valid values is sufficiently small.	Examples of discrete possibilities without numeric content are days of the week, months of the year, make and model of car. An example of numeric content that would be an exception is the number of connected devices within a finite number of connections. It's better for a user to pick a value of 0, 1, 2, 3 from a ring control than to instead have a numeric control and need to make sure the specified value is in

Guideline	Required or Recommended?	Details	Example(s)
			range.
If you use text rings for numeric content, do not create an item with text that represents a different numeric value.	Recommended	N/A	For example, a selection of 2 assigned to the value 0 could confuse the user.
Use the timestamp data type for time values in your API.	Recommended	N/A	N/A
Use a double-precision value to specify the number of seconds for timeout values in your API. Use -1 to indicate that the diagram object has no timeout limit.	Recommended	N/A	N/A
Use the waveform data type when your VI acquires or generates analog waveforms.	Recommended	N/A	N/A
If your VI has a string input or output, consider whether another data type, such as an enum,	Recommended	N/A	If the user needs to pick a day of the week, an enum with the seven days in it is easier to use than typing the name of the day as a string, because of possible problems with the case,

Guideline	Required or Recommended?	Details	Example(s)
might make your API easier to use.			spelling, and abbreviation.
If you use a cluster as an input or output of your VI, make sure the cluster logically groups parameters.	Recommended	Do not use clusters only to reduce the number of inputs and outputs on your VI, because clustering things unnecessarily to reduce the number of inputs to the VI is not a logical organization.	N/A

Related concepts:

- Best Practices for Designing and Developing an Application Programming Interface (API) in G Web Development Software

Palette Taxonomy for Distributed APIs

Refer to the following table for best practices for palette taxonomy.

Guideline	Required or Recommended?	Details	Example(s)
Make the root palette easy to browse.	Recommended	Seven to 10 categories allows for full-size icons in a single-column layout.	N/A
Design a structure that can expand beyond the original size of the root palette for nodes you	Recommended	N/A	N/A

Guideline	Required or Recommended?	Details	Example(s)
may develop in the future.			
Make palettes for similar categories consistent.	Recommended	N/A	Across different palettes, such as numeric, string, file, and so on, that have a constants palette, similarly organize all the constants palettes.
Add synonyms or keywords to your palettes to improve the findability of your VIs.	Recommended	N/A	Mathematicians commonly refer to the functionality of the Quotient and Remainder node as ***mod***. Make sure ***mod*** is a keyword for Quotient and Remainder so that a search for ***mod*** returns that node.

Related concepts:

- Best Practices for Designing and Developing an Application Programming Interface (API) in G Web Development Software

Documentation for Distributed APIs

Refer to the following table for best practices for documenting an API.

Guideline	Required or Recommended?	Details
Review the VI description and parameter descriptions of all public VIs in your API for correct meaning for	Required	N/A

Guideline	Required or Recommended?	Details
what the VI does, usability by third parties who are unfamiliar with your API, spelling, grammar, and naming conventions for G Web Development Software.		

Related concepts:

- Best Practices for Designing and Developing an Application Programming Interface (API) in G Web Development Software

Error Message Design for Distributed APIs

Refer to the following table for best practices for designing error messages.

Guideline	Required or Recommended?	Details
Reserve a range of error codes for your API and use those codes for error conditions specific to your API.	Required	Pick a range for error codes between -8999 through -8000, 5000 through 9999, and 500,000 through 599,999, which are reserved for external users.
When generating or manipulating errors on the diagram of your API VIs, make sure you use the Error API to construct and manipulate error clusters.	Required	Never bundle or unbundle error cluster elements directly in G Web Development Software.
If the VI has no way to generate errors, consider excluding error inputs and outputs.	Recommended	If a VI does not generate errors, users of the VI can take advantage of parallelism in G Web Development Software.

Guideline	Required or Recommended?	Details
If you exclude error inputs and outputs on a VI in your API, leave the error input and output locations on the connector pane empty.	Recommended	When unused error inputs and outputs on a VI are left empty, you can add error handling to the VI later.

Related concepts:

- Best Practices for Designing and Developing an Application Programming Interface (API) in G Web Development Software

API Design for Distributed APIs

Refer to the following table for best practices for designing VIs.

Guideline	Required or Recommended?	Details	Example(s)
Include every public VI in your API in the API palette.	Required	Users may not look on disk to find advanced VIs. If a VI is not ready or intended for public consumption, make the VI private in the API component.	N/A
Do not use a VI in your API to open a dialog box unless that is the stated intent of the VI.	Required	N/A	N/A
If you expect multiple	Recommended	If the VI stores any internal state in feedback nodes or	N/A

Guideline	Required or Recommended?	Details	Example(s)
calls of your API VI to run in parallel, consider making the VI reentrant.		uninitialized shift registers, make the VI reentrant and ***stateful***, such as a preallocated clone. If the VI doesn't store any internal state, make it ***stateless***, as a shared clone.	

Related concepts:

- Best Practices for Designing and Developing an Application Programming Interface (API) in G Web Development Software

Interfaces for MATLAB

An ***Interface for MATLAB*** (`.mli`) is a document in which you define calls to a MATLAB file (`.m` or `.mlx`) in your G dataflow application.

MATLAB files can be either ***functions*** or ***scripts***. NI recommends that you format MATLAB programs into functions because functions perform better than scripts and offer a clean programming model.

In an Interface for MATLAB, you create ***interface nodes*** that map to arguments in a MATLAB function or variables in a MATLAB script. Visual representations of interface nodes appear on the **Project Item** » **Software** palette on the diagram. You can place and wire interface nodes in your application.

When you execute the application, the Interface for MATLAB invokes MATLAB, which calls the MATLAB file. Input data passes from the diagram to MATLAB, and data returns from MATLAB to the diagram.

The Interface for MATLAB supports Windows targets only.

Examples

Search within the programming environment to access the following installed examples:

- ***Interface for MATLAB Fundamentals***
- ***Interface for MATLAB Working with nD Arrays***
- ***Interface for MATLAB Working with Structures***
- ***MATLAB User Defined Function***
- ***MATLAB Monte Carlo Calculation***
- ***Prime Number Calculation***

MATLAB® is a registered trademark of The MathWorks, Inc.

Related tasks:

- Calling MATLAB Functions and Scripts
- Debugging MATLAB Functions and Scripts
- Migrating from MathScript Node to Interface for MATLAB

Calling MATLAB Functions and Scripts

Create an Interface for MATLAB to define calls to a MATLAB function or script.

1. On the Project Files tab, select **New » Interface for MATLAB®**.
2. In the Interface for MATLAB, use one of the following options to specify a function or script you want to call:
   - Click the **...** button and select a MATLAB file on disk.
   - Enter the name of a MATLAB function or the filename of a MATLAB file on the MATLAB search path. Use this option to build your G dataflow application into an executable (`.exe`).
3. On the Document tab, choose the **File type** of the MATLAB file you want to call.

   NI recommends that you format MATLAB programs into functions because functions perform better than scripts and offer a clean programming model.

4. Define the ***interface node*** that maps to the arguments in the function or variables in the script.
   a. Click **Add interface node**.
   b. On the Item tab, enter a name for the interface node.
      The name will show up in the node icon.
   c. Click **Add parameter**.
   d. On the Item tab, specify the parameter name, data type, and behavior.
   e. Add more parameters as necessary.
      The number of parameters in the interface node must match the number of arguments in the function or the number of variables in the script.
5. Save the Interface for MATLAB.
6. Open the VI in which you want to call the MATLAB function or script.
7. On the diagram palette, click **Project Items** » **Software** to find the interface node you defined in the Interface for MATLAB.
8. Drop the interface node on the diagram.
9. Wire the interface node and complete the diagram.
10. Run the VI.

    The MATLAB Command Window automatically launches. Input data passes from the diagram to MATLAB, and data returns from MATLAB to the diagram.

> **Note** If you have multiple versions of MATLAB installed, by default the Interface for MATLAB invokes the version of MATLAB you most recently installed.

Related concepts:

- Interfaces for MATLAB

Related tasks:

- Debugging MATLAB Functions and Scripts

Related information:

- Programming Scripts and Functions in MATLAB

Debugging MATLAB Functions and Scripts

You can debug MATLAB functions and scripts when you use an Interface for MATLAB to call them in your G dataflow application.

Complete the following steps to debug a function or script.

1. In an Interface for MATLAB, ensure that you specify the correct file path to a MATLAB function or script.
2. Click **Open in MATLAB®** to open the function or script in the MATLAB Editor.
3. In the MATLAB Editor that launches, add a breakpoint on the line of code where you think the problem could be.
4. Run your G dataflow application.
   The execution of the MATLAB function or script pauses at the specific line where you add the breakpoint.
5. While the function or script is paused, you can view the value of each argument in the MATLAB Editor.
6. Click **Continue** in the MATLAB Editor.
   The G dataflow application finishes executing the remaining code.
7. Change the values of arguments or modify the script or function to produce expected results by using standard MATLAB functionality.

Importing and Exporting MATLAB Data

In a G dataflow application, you can import data from or export data to MATLAB® using MATLAB formatted binary files (`.mat`).

To import data from MATLAB, click the **Import** button in the Captured Data tab and select a `.mat` file. You can use the data in your G dataflow application.

To export data to MATLAB, right-click a data item in the Captured Data tab and select **Export** to export to a `.mat` file. You can then load the data file in MATLAB to analyze the data.

Related information:

- Capturing and Analyzing Data

Migrating from MathScript Node to Interface for MATLAB

Use an ***Interface for MATLAB*** to migrate source code that contains a MathScript Node.

When you open a VI containing a MathScript Node, the MathScript Node is replaced with a Sequence Structure and the original MathScript code is converted to a MATLAB file (`.m`). The VI is broken and you must modify the code to replicate the behavior of the original code.

Complete the following steps to migrate your code:

1. Ensure that the generated MATLAB file works in MATLAB.

   Refer to the comments in the Sequence Structure to locate the MATLAB file on disk. If the MATLAB file contains MathScript-specific function names, you must modify the file to use MATLAB function names. Refer to Migrating MathScript Functions to MathWorks® Functions for more information about mapping MathScript function names and their corresponding function names in MathWorks products.

2. Create an Interface for MATLAB to call the MATLAB file.

   > **Note**
   > - You must choose **Function** as the **File type** on the Document tab because the generated MATLAB file is a function.
   > - You must configure parameters of the interface node using consistent input/output arguments in the MATLAB file. Find input/output arguments of the MATLAB file from comments in the Sequence Structure.

3. Return to the VI diagram and rewire controls and indicators to the interface node.
   a. Click **Project Files** » **Software**, select the interface node, and drop the interface node on the diagram.
   b. Wire controls and indicators to the interface node by replicating wire connections to the Sequence Structure.
   c. Remove the Sequence Structure and clean up the diagram.
   d. Save the VI.

Notice that the VI is no longer broken. The modified code has the same behavior as the original code.

Migrating MathScript Functions to MathWorks Functions

The following table lists MathScript function names and their corresponding function names in MathWorks products. Review the usage of these functions when you migrate from MathScript Node to Interface for MATLAB.

MathScript Function Name	MathWorks Function Name	MathWorks Product
ac_to_poly	ac2poly	Signal Processing Toolbox™
ac_to_rc	ac2rc	Signal Processing Toolbox™
ac_to_rcschur	schurrc	Signal Processing Toolbox™
accumproducts	cumprod	MATLAB®
accumsums	cumsum	MATLAB®
accumtrapint	cumtrapz	MATLAB®
ackermann	acker	Control System Toolbox™
add_noise	imnoise	Image Processing Toolbox™
algriccati	care	Control System Toolbox™
ar_burg	arburg	Signal Processing Toolbox™
ar_covar	arcov	Signal Processing Toolbox™
ar_mcovar	armcov	Signal Processing Toolbox™
ar_yule	aryule	Signal Processing Toolbox™
arginchk	narginchk	MATLAB®
arginnum	nargin	MATLAB®
argoutchk	nargoutchk	MATLAB®
argoutnum	nargout	MATLAB®
augmentstate	augstate	Control System Toolbox™
balance_diag	ssbal	Control System Toolbox™

MathScript Function Name	MathWorks Function Name	MathWorks Product
balance_grammian	balreal	Control System Toolbox™
barhoriz	barh	MATLAB®
base_to_dec	base2dec	MATLAB®
bessel_h	besselh	MATLAB®
bessel_i	besseli	MATLAB®
bessel_j	besselj	MATLAB®
bessel_k	besselk	MATLAB®
bessel_y	bessely	MATLAB®
beta_incomplete	betainc	MATLAB®
beta_ln	betaln	MATLAB®
bin_to_dec	bin2dec	MATLAB®
bitnot	bitcmp	MATLAB®
bitreverseorder	bitrevorder	Signal Processing Toolbox™
blockdiag	blkdiag	MATLAB®
buffermx	buffer	Signal Processing Toolbox™
c_to_d	c2d	Control System Toolbox™
canonical	canon	Control System Toolbox™
cart_to_polar	cart2pol	MATLAB®
cart_to_sphere	cart2sph	MATLAB®
ccepstrum	cceps	Signal Processing Toolbox™
cctranspose	ctranspose	MATLAB®
char	setstr	MATLAB®
chirpzt	czt	Signal Processing Toolbox™
circularshift	circshift	MATLAB®
clfig	clf	MATLAB®
clgraph	clf	MATLAB®

MathScript Function Name	MathWorks Function Name	MathWorks Product
clout	clc	MATLAB®
coherence	cohere	Signal Processing Toolbox™
coherence_ms	mscohere	Signal Processing Toolbox™
colormapplot	rgbplot	MATLAB®
companion	compan	MATLAB®
condeign	condeig	MATLAB®
condestimate	condest	MATLAB®
condrecip	rcond	MATLAB®
conjugate	conj	MATLAB®
contouris	contourf	MATLAB®
contours	contourc	MATLAB®
conv2d	conv2	MATLAB®
convcirc	cconv	Signal Processing Toolbox™
convexhull	convhull	MATLAB®
convmx	convmtx	Signal Processing Toolbox™
corrcoeff	corrcoef	MATLAB®
corrmx	corrmtx	Signal Processing Toolbox™
covarmx	cov	MATLAB®
crosscorr	xcorr	Signal Processing Toolbox™
crosscorr2d	xcorr2	Signal Processing Toolbox™
crosscovar	xcov	Signal Processing Toolbox™
crosspsd	cpsd	Signal Processing Toolbox™
crosssd	csd	Signal Processing Toolbox™
ctrbmx	ctrb	Control System Toolbox™
ctrbstairs	ctrbf	Control System Toolbox™
d_to_c	d2c	Control System Toolbox™

MathScript Function Name	MathWorks Function Name	MathWorks Product
d_to_d	d2d	Control System Toolbox™
dalgriccati	dare	Control System Toolbox™
datatype	class	MATLAB®
date_to_num	datenum	MATLAB®
date_to_str	datestr	MATLAB®
date_to_vector	datevec	MATLAB®
datescale	datetick	MATLAB®
dec_to_base	dec2base	MATLAB®
dec_to_bin	dec2bin	MATLAB®
dec_to_hex	dec2hex	MATLAB®
deflate	squeeze	MATLAB®
deg_to_rad	deg2rad	MATLAB®
delay_to_z	delay2z	Control System Toolbox™
density_kernel	ksdensity	Statistics and Machine Learning Toolbox™
dftmx	dftmtx	Signal Processing Toolbox™
difference	diff	MATLAB®
digitreverseorder	digitrevorder	Signal Processing Toolbox™
dirichlet	diric	Signal Processing Toolbox™
dlaplacian	del2	MATLAB®
dlqr_y	dlqry	Control System Toolbox™
dlyapunov	dlyap	Control System Toolbox™
drandss	drss	Control System Toolbox™
drandtf	tf	Control System Toolbox™
drandzpk	zpk	Control System Toolbox™
duplicate	deal	MATLAB®
eigsort	eigs	MATLAB®

MathScript Function Name	MathWorks Function Name	MathWorks Product
elliptic_int	ellipke	MATLAB®
elliptic_j	ellipj	MATLAB®
eqtflen	eqtflength	Signal Processing Toolbox™
erf_inv	erfinv	MATLAB®
erfc_inv	erfcinv	MATLAB®
erfc_scale	erfcx	MATLAB®
estimator	estim	Control System Toolbox™
evalfreq	evalfr	Control System Toolbox™
exp_int	expint	MATLAB®
expmx	expm	MATLAB®
expmx_eign	expmdemo3	MATLAB®
expmx_pade	expmdemo1	MATLAB®
expmx_taylor	expmdemo2	MATLAB®
eyediagram	eyediagram	Communications Toolbox™
fft2d	fft2	MATLAB®
filter_2d	filter2	MATLAB®
filter_fft	fftfilt	Signal Processing Toolbox™
filter_impulse	impinvar	Signal Processing Toolbox™
filter_lattice	latcfilt	Signal Processing Toolbox™
filter_median	medfilt1	Signal Processing Toolbox™
filter_rcos	rcosflt	Communications Toolbox™
filter_sg	sgolayfilt	Signal Processing Toolbox™
filter_sos	sosfilt	Signal Processing Toolbox™
filter_zerophase	filtfilt	Signal Processing Toolbox™
filteric	filtic	Signal Processing Toolbox™
findnz	find	MATLAB®

MathScript Function Name	MathWorks Function Name	MathWorks Product
fir_fs	fir2	Signal Processing Toolbox™
fir_gauss	firgauss	Signal Processing Toolbox™
fir_gaussps	gaussfir	Signal Processing Toolbox™
fir_interp	intfilt	Signal Processing Toolbox™
fir_lsq	firls	Signal Processing Toolbox™
fir_pm	firpm	Signal Processing Toolbox™
fir_pmord	firpmord	Signal Processing Toolbox™
fir_rcos	firrcos	Signal Processing Toolbox™
fir_remez	remez	Signal Processing Toolbox™
fir_remezord	remezord	Signal Processing Toolbox™
fir_sgsmooth	sgolay	Signal Processing Toolbox™
fir_win	fir1	Signal Processing Toolbox™
flatindex	sub2ind	MATLAB®
fmin_bracket	fminbnd	MATLAB®
fmin_lp	linprog	Optimization Toolbox™
fmin_qp	quadprog	Optimization Toolbox™
fread_audio	audioread	MATLAB®
freq_space	freqspace	MATLAB®
freqsd	freqs	Signal Processing Toolbox™
freqzd	freqz	Signal Processing Toolbox™
funmx	funm	MATLAB®
gamma_incomplete	gammainc	MATLAB®
gamma_ln	gammaln	MATLAB®
gaussmonopulse	gmonopuls	Signal Processing Toolbox™
gausspulse	gauspuls	Signal Processing Toolbox™
gensignal	gensig	Control System Toolbox™

MathScript Function Name	MathWorks Function Name	MathWorks Product
grammian	gram	Control System Toolbox™
hconcatmx	horzcat	MATLAB®
hessenberg	hess	MATLAB®
hex_to_dec	hex2dec	MATLAB®
hex_to_num	hex2num	MATLAB®
hilbertmx	hilb	MATLAB®
histogram	hist	MATLAB®
histogramc	histc	MATLAB®
iccepstrum	icceps	Signal Processing Toolbox™
ifft2d	ifft2	MATLAB®
ifft_shift	ifftshift	MATLAB®
iir_bessel	besself	Signal Processing Toolbox™
iir_besselzpk	besselap	Signal Processing Toolbox™
iir_butter	butter	Signal Processing Toolbox™
iir_butterord	buttord	Signal Processing Toolbox™
iir_butterzpk	buttap	Signal Processing Toolbox™
iir_cheby1	cheby1	Signal Processing Toolbox™
iir_cheby1ord	cheb1ord	Signal Processing Toolbox™
iir_cheby1zpk	cheb1ap	Signal Processing Toolbox™
iir_cheby2	cheby2	Signal Processing Toolbox™
iir_cheby2ord	cheb2ord	Signal Processing Toolbox™
iir_cheby2zpk	cheb2ap	Signal Processing Toolbox™
iir_elliptic	ellip	Signal Processing Toolbox™
iir_ellipticord	ellipord	Signal Processing Toolbox™
iir_ellipticzpk	ellipap	Signal Processing Toolbox™
iir_maxflat	maxflat	Signal Processing Toolbox™

MathScript Function Name	MathWorks Function Name	MathWorks Product
iir_steigmcbride	stmcb	Signal Processing Toolbox™
iir_yulewalker	yulewalk	Signal Processing Toolbox™
imagescaled	imagesc	MATLAB®
impzd	impz	Signal Processing Toolbox™
ind_to_sub	ind2sub	MATLAB®
int_to_str	int2str	MATLAB®
interpolate	interp	Control System Toolbox™
interpolate1d	interp1	MATLAB®
interpolate2d	interp2	MATLAB®
interpolateft	interpft	MATLAB®
intrp2d_uneven	griddata	MATLAB®
invfreqsd	invfreqs	Signal Processing Toolbox™
invfreqzd	invfreqz	Signal Processing Toolbox™
invhilbertmx	invhilb	MATLAB®
iopzgraph	iopzmap	Control System Toolbox™
is_char	ischar	MATLAB®
is_dir	isdir	MATLAB®
is_empty	isempty	MATLAB®
is_equal	isequal	MATLAB®
is_equalnan	isequalwithequalnans	MATLAB®
is_field	isfield	MATLAB®
is_finite	isfinite	MATLAB®
is_hold	ishold	MATLAB®
is_inf	isinf	MATLAB®
is_inpolygon	inpolygon	MATLAB®
is_keyword	iskeyword	MATLAB®

MathScript Function Name	MathWorks Function Name	MathWorks Product
is_letter	isletter	MATLAB®
is_logical	islogical	MATLAB®
is_membermx	ismember	MATLAB®
is_nan	isnan	MATLAB®
is_numeric	isnumeric	MATLAB®
is_prime	isprime	MATLAB®
is_real	isreal	MATLAB®
is_scalar	isscalar	MATLAB®
is_sorted	issorted	MATLAB®
is_space	isspace	MATLAB®
is_string	isstr	MATLAB®
is_struct	isstruct	MATLAB®
is_student	isstudent	MATLAB®
is_to_rc	is2rc	Signal Processing Toolbox™
is_validvarname	isvarname	MATLAB®
kaiserwinord	kaiserord	Signal Processing Toolbox™
kalman_d	kalmd	Control System Toolbox™
lar_to_rc	lar2rc	Signal Processing Toolbox™
lattice_to_tf	latc2tf	Signal Processing Toolbox™
leftdiv	ldivide	MATLAB®
leftdivmx	mldivide	MATLAB®
lib_call	calllib	MATLAB®
lib_funclist	libfunctionsview	MATLAB®
lib_isloaded	libisloaded	MATLAB®
lib_load	loadlibrary	MATLAB®
lib_unload	unloadlibrary	MATLAB®

MathScript Function Name	MathWorks Function Name	MathWorks Product
linearsolve	linsolve	MATLAB®
linramp	linspace	MATLAB®
logmx	logm	MATLAB®
logramp	logspace	MATLAB®
lowercase	lower	MATLAB®
lp_to_bp	lp2bp	Signal Processing Toolbox™
lp_to_bs	lp2bs	Signal Processing Toolbox™
lp_to_hp	lp2hp	Signal Processing Toolbox™
lp_to_lp	lp2lp	Signal Processing Toolbox™
lqr_d	lqrd	Control System Toolbox™
lqr_dy	ss	Control System Toolbox™
lqr_y	lqry	Control System Toolbox™
lsf_to_poly	lsf2poly	Signal Processing Toolbox™
lyapunov	lyap	Control System Toolbox™
margins	allmargin	Control System Toolbox™
maxfloat	realmax	MATLAB®
maxfloatint	flintmax	MATLAB®
maxnamelen	namelengthmax	MATLAB®
meshgrid2d	meshgrid	MATLAB®
minfloat	realmin	MATLAB®
minimal	minreal	Control System Toolbox™
minimal_state	sminreal	Control System Toolbox™
minrepseq	seqperiod	Signal Processing Toolbox™
minus1	uminus	MATLAB®
mirror	flipdim	MATLAB®
mirrorh	fliplr	MATLAB®

MathScript Function Name	MathWorks Function Name	MathWorks Product
mirrorv	flipud	MATLAB®
moment_central	moment	Statistics and Machine Learning Toolbox™
mreduce	modred	Control System Toolbox™
multmx	mtimes	MATLAB®
mx_to_str	mat2str	MATLAB®
nextpowerof2	nextpow2	MATLAB®
normestimate	normest	MATLAB®
num_to_str	num2str	MATLAB®
numdays	eomday	MATLAB®
numdims	ndims	MATLAB®
numelements	numel	MATLAB®
numnz	nnz	MATLAB®
nz	nonzeros	MATLAB®
obsvmx	obsv	Control System Toolbox™
obsvstair	obsvf	Control System Toolbox™
ode_adams	ode113	MATLAB®
ode_bdf15	ode15s	MATLAB®
ode_bdf23	ode23tb	MATLAB®
ode_rk23	ode23	MATLAB®
ode_rk45	ode45	MATLAB®
ode_rosen	ode23s	MATLAB®
odepset	odeset	MATLAB®
padm	pamdemod	Communications Toolbox™
pam	pammod	Communications Toolbox™
peakfcn1d	humps	MATLAB®
peakfcn2d	peaks	MATLAB®

MathScript Function Name	MathWorks Function Name	MathWorks Product
phasezd	phasez	Signal Processing Toolbox™
plotcoord	gplot	MATLAB®
plottext	gtext	MATLAB®
plus1	uplus	MATLAB®
polar_to_cart	pol2cart	MATLAB®
poleplace	place	Control System Toolbox™
poles	pole	Control System Toolbox™
poly_pw	mkpp	MATLAB®
poly_pwhermite	pchip	MATLAB®
poly_scale	polyscale	Signal Processing Toolbox™
poly_stable	polystab	Signal Processing Toolbox™
poly_to_ac	poly2ac	Signal Processing Toolbox™
poly_to_lsf	poly2lsf	Signal Processing Toolbox™
poly_to_rc	poly2rc	Signal Processing Toolbox™
polyderivative	polyder	MATLAB®
polyeign	polyeig	MATLAB®
polygonarea	polyarea	MATLAB®
polyintegral	polyint	MATLAB®
polyvalmx	polyvalm	MATLAB®
powermx	mpower	MATLAB®
powerof2	pow2	MATLAB®
powerofreal	realpow	MATLAB®
psd_burg	pburg	Signal Processing Toolbox™
psd_covar	pcov	Signal Processing Toolbox™
psd_mcovar	pmcov	Signal Processing Toolbox™
psd_periodogram	periodogram	Signal Processing Toolbox™

MathScript Function Name	MathWorks Function Name	MathWorks Product
psd_welch	pwelch	Signal Processing Toolbox™
psd_yule	pyulear	Signal Processing Toolbox™
pspec_eign	peig	Signal Processing Toolbox™
pspec_music	pmusic	Signal Processing Toolbox™
pulsetrain	pulstran	Signal Processing Toolbox™
pzgraph	pzmap	Control System Toolbox™
qadm	qamdemod	Communications Toolbox™
qam	qammod	Communications Toolbox™
quadn_trap	trapz	MATLAB®
quantdecode	udecode	Signal Processing Toolbox™
quantencode	uencode	Signal Processing Toolbox™
rad_to_deg	rad2deg	MATLAB®
randnormal	randn	MATLAB®
randpermutation	randperm	MATLAB®
randss	rss	Control System Toolbox™
randtf	tf	Control System Toolbox™
randzpk	zpk	Control System Toolbox™
rc_to_ac	rc2ac	Signal Processing Toolbox™
rc_to_is	rc2is	Signal Processing Toolbox™
rc_to_lar	rc2lar	Signal Processing Toolbox™
rc_to_poly	rc2poly	Signal Processing Toolbox™
rcepstrum	rceps	Signal Processing Toolbox™
rcos	rcosine	Communications Toolbox™
rectintarea	rectint	MATLAB®
rectpulse	rectpuls	Signal Processing Toolbox™
ref_plotarea	gca	MATLAB®

MathScript Function Name	MathWorks Function Name	MathWorks Product
ref_plotwin	gcf	MATLAB®
regex	regexp	MATLAB®
regex_convert	regexptranslate	MATLAB®
regex_i	regexpi	MATLAB®
regex_replace	regexprep	MATLAB®
regulator	reg	Control System Toolbox™
remove_field	rmfield	MATLAB®
reorderdim	permute	MATLAB®
reorderdiminv	ipermute	MATLAB®
repeatmx	repmat	MATLAB®
resample_fir	upfirdn	Signal Processing Toolbox™
reshapemx	reshape	MATLAB®
residuezd	residuez	Signal Processing Toolbox™
reverse_vector	flip	MATLAB®
revlevinson	rlevinson	Signal Processing Toolbox™
rgb_to_grayscale	rgb2gray	MATLAB®
rightdiv	rdivide	MATLAB®
rightdivmx	mrdivide	MATLAB®
rlocusfind	rlocfind	Control System Toolbox™
root_eign	rooteig	Signal Processing Toolbox™
root_music	rootmusic	Signal Processing Toolbox™
rotate90	rot90	MATLAB®
rotateplane	planerot	MATLAB®
scatter3d	scatter3	MATLAB®
selectdata	ginput	MATLAB®
semilog_x	semilogx	MATLAB®

MathScript Function Name	MathWorks Function Name	MathWorks Product
semilog_y	semilogy	MATLAB®
sets_diff	setdiff	MATLAB®
sets_intersect	intersect	MATLAB®
sets_union	union	MATLAB®
sets_unique	unique	MATLAB®
sets_xor	setxor	MATLAB®
showplot	shg	MATLAB®
singularvalues	sigma	Control System Toolbox™
sortconjugate	cplxpair	MATLAB®
sortdown	dsort	Control System Toolbox™
sortdownreal	esort	Control System Toolbox™
sos_to_ss	sos2ss	Signal Processing Toolbox™
sos_to_tf	sos2tf	Signal Processing Toolbox™
sos_to_zpk	sos2zp	Signal Processing Toolbox™
soundscaled	soundsc	MATLAB®
spectrogram	specgram	Signal Processing Toolbox™
sphere_to_cart	sph2cart	MATLAB®
splinefit	spline	MATLAB®
sqrtmx	sqrtm	MATLAB®
sqrtofreal	realsqrt	MATLAB®
ss_to_sos	ss2sos	Signal Processing Toolbox™
ss_to_ss	ss2ss	Control System Toolbox™
ss_to_tf	ss2tf	Control System Toolbox™
ss_to_zpk	ss2zp	Control System Toolbox™
stem3d	stem3	MATLAB®
stepsignal	stepfun	Control System Toolbox™

MathScript Function Name	MathWorks Function Name	MathWorks Product
stepzd	stepz	Signal Processing Toolbox™
str_to_double	str2double	MATLAB®
str_to_mx	str2mat	MATLAB®
str_to_num	str2num	MATLAB®
strcmp_i	strcmpi	MATLAB®
strcmp_n	strncmp	MATLAB®
strcmp_ni	strncmpi	MATLAB®
strconcat	strcat	MATLAB®
strfindall	strfind	MATLAB®
stripplot	strips	Signal Processing Toolbox™
strjustify	strjust	MATLAB®
strmatchall	strmatch	MATLAB®
strreplace	strrep	MATLAB®
strtoken	strtok	MATLAB®
strtrimwhite	strtrim	MATLAB®
strvconcat	strvcat	MATLAB®
strvectorize	vectorize	MATLAB®
subspaceangle	subspace	MATLAB®
surface	surf	MATLAB®
surfacecontour	surfc	MATLAB®
surfacenorm	surfnorm	MATLAB®
sys_filter	filt	Control System Toolbox™
sys_order1	rss	Control System Toolbox™
sys_order2	rss	Control System Toolbox™
tf_estimate	tfe	Signal Processing Toolbox™
tf_estimateplot	tfestimate	Signal Processing Toolbox™

MathScript Function Name	MathWorks Function Name	MathWorks Product
tf_to_lattice	tf2latc	Signal Processing Toolbox™
tf_to_sos	tf2sos	Signal Processing Toolbox™
tf_to_ss	tf2ss	Control System Toolbox™
tf_to_zpk	tf2zp	Control System Toolbox™
tf_to_zpk_eqlen	tf2zpk	Signal Processing Toolbox™
timerstart	tic	MATLAB®
timerstop	toc	MATLAB®
titles	title	MATLAB®
triangsearch	isinterior	MATLAB®
tripulse	tripuls	Signal Processing Toolbox™
unwrapphase	unwrap	MATLAB®
updatechol	cholupdate	MATLAB®
uppercase	upper	MATLAB®
vandermonde	vander	MATLAB®
vconcatmx	vertcat	MATLAB®
view_image	imshow	MATLAB®
who_all	whos	MATLAB®
win_bartlett	bartlett	Signal Processing Toolbox™
win_bartletthann	barthannwin	Signal Processing Toolbox™
win_blackman	blackman	Signal Processing Toolbox™
win_blackmanharris	blackmanharris	Signal Processing Toolbox™
win_bohman	bohmanwin	Signal Processing Toolbox™
win_cheby	chebwin	Signal Processing Toolbox™
win_flattop	flattopwin	Signal Processing Toolbox™
win_gauss	gausswin	Signal Processing Toolbox™
win_hamming	hamming	Signal Processing Toolbox™

MathScript Function Name	MathWorks Function Name	MathWorks Product
win_hann	hann	Signal Processing Toolbox™
win_hann2	hanning	Signal Processing Toolbox™
win_kaiser	kaiser	Signal Processing Toolbox™
win_nuttall	nuttallwin	Signal Processing Toolbox™
win_parzen	parzenwin	Signal Processing Toolbox™
win_rect	rectwin	Signal Processing Toolbox™
win_taylor	taylorwin	Signal Processing Toolbox™
win_triangular	triang	Signal Processing Toolbox™
win_tukey	tukeywin	Signal Processing Toolbox™
xlabels	xlabel	MATLAB®
xlimit	xlim	MATLAB®
ylabels	ylabel	MATLAB®
ylimit	ylim	MATLAB®
zlimit	zlim	MATLAB®
zpk_to_sos	zp2sos	Signal Processing Toolbox™
zpk_to_ss	zp2ss	Control System Toolbox™
zpk_to_tf	zp2tf	Control System Toolbox™

Creating User Interfaces

To learn more about creating user interfaces, such as how to center cursors on graphs or charts, how to clear indicator display data, how to set the tabbing order, and how to write multiple plots to a graph or chart, click the topics in the left-hand navigation.

Centering a Cursor on a Graph or Chart

If you cannot locate a cursor that you added to your graph or chart, you can reset the cursor position to the center of your graph or chart. Use the cursor legend to center a

cursor.

Note Cursors are not available for intensity graphs.

1. If you do not see the cursor legend, select **Cursor Legend** in the **Parts** section on the Item tab for the graph or chart.
2. On the **Cursor Legend**, navigate to the cursor you want to center.
3. Ensure the **Visible?** button is selected.
4. Click the **Center Cursor** button to center the cursor.

Clearing Indicator Display Data in a Chart, Graph, or Array

Right-click the chart, graph, or array and select **Clear Data** to remove all data from the indicator displays.

Setting the Tabbing Order for Controls on the Panel

Controls on the panel have an order, called tabbing order, that is unrelated to their position on the panel.

Tabbing order is based on the order in which you place controls on the panel. The first control you create on the panel is element 0, the second is 1, and so on. If you delete a control, the tabbing order adjusts automatically. The tabbing order determines the order in which the software selects controls when the user presses the <Tab> key while a VI runs.

Use the **Tab Order** on the Document tab to configure tabbing order for controls on the panel.

Writing Multiple Plots to a Graph or Chart

Before you write multiple plots to a graph or chart, you must generate all the data sets you want to plot. Make sure each set of data has the same data type.

1. Wire each set of data you want to plot to a Build Array node.

2. Wire the `appended array` output from Build Array to a graph or chart indicator. If your data consists of numeric, complex, or cluster values, `appended array` is a 2D array. Each row of the array is a separate plot. If your data consists of waveforms, `appended array` is a 1D array. Each waveform is a separate plot.

Testing and Debugging

When the results of your application are not what you expect, use a set of tools to determine where errors occur within your code.

Although errors are often detected automatically, sometimes your code can run successfully but not as intended. When this happens, you need to identify the source of the unintended behaviors.

The following debugging tools can help you in this process:

- Probes
- Breakpoints

Highlighting Execution of the Diagram

Execution highlighting reduces the speed of execution and displays data bubbles that move along the wires, revealing the data that each node receives when it executes.

When you begin debugging a program, you may not be able to identify which part of the code introduces the incorrect behavior. To gain a better idea of how data progresses and changes as it flows through the code, you can use execution highlighting.

Execution highlighting helps you detect the following kinds of unintended behavior:

- While Loops that never terminate
- Case Structures that execute an unexpected case
- Data values that do not match the expected value

You can turn on execution highlighting by clicking the **Highlight execution** button on the document toolbar.

Using Probes to Check Values on a Wire

When your WebVI is running in the editor environment, you may want to check the values on the wires to determine if and where any unexpected data occurs.

To place a probe on a wire, right-click on the wire and select **Add Probe**. A probe displays the data from the wire it is on. To view a probe in a temporary overlay on the diagram, hover over the probe marker on a wire. If desired, you can then click the **Pin** button to keep this probe visible. From the **Options** drop-down you can choose the different display styles and also remove the probe, as required.

You can also find probes in the Debugging tab on the left side of the editor environment. The order in the list correlates with the numbers in the probe markers on the wires.

Probes display the most recent value carried by a wire. You can hover over the probes in the list and observe the tip strip that specifies when the value was last updated.

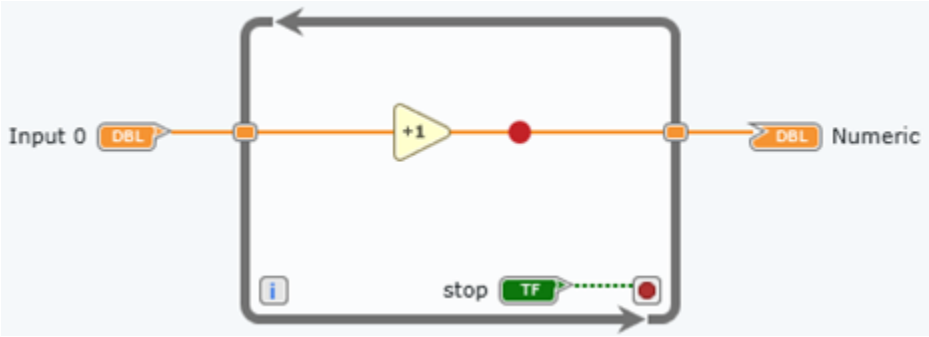
The values displayed in the probe list are limited to a line of text. However, you can hover over a probe and click the **>** button to display an expanded view of the probe and the **Options** drop-down.

Pausing Execution with Breakpoints

When looking for a problem in your code, you may have an idea of the general area where the problem exists. To help focus on this area, you can use a breakpoint to pause the VI at a specified point in the program.

When the VI reaches a breakpoint during execution, you can take the following actions:

- Single-step through execution using the single-stepping buttons.
- Check intermediate values on probes that you placed on wires prior to running the VI.
- Change values of panel controls.
- Click the **Resume** button to continue running to the next breakpoint or until the VI finishes running.



You can add a breakpoint to any wire or node in the code, or in a row of a text-based programming node. Add a breakpoint by right-clicking the wire, node, or row and selecting **Add breakpoint**.

You can add conditional logic to a breakpoint to pause execution if a condition is met. Add a condition by right clicking the breakpoint and selecting **Pause on condition**. Configure the conditions of the breakpoint in the Debugging pane.

Single-Stepping through VIs

While execution highlighting slows the execution of your code, single-stepping allows you to have more control of viewing individual actions of the program.

With execution highlighting, execution slows down, and the code executes until completion. With single-stepping, you can execute a single node at a time, causing the program to pause after the node completes.

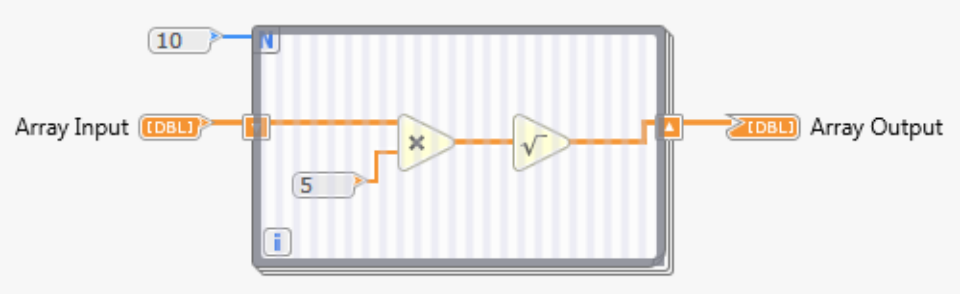
You can use single-stepping in three ways: **Step In**, **Step Out**, and **Step Over**. You can find these options in the Debugging tab.

	***Step In***—Display the code and pause execution if a node is a subVI and represents more code.

	For a node that you cannot open, use this option to highlight the node and pause its execution. For text-based programming nodes, Step-In processes the code row-wise.
	***Step Out***— Complete the execution of the current diagram or subdiagram and pause.
	***Step Over***—Execute a node without stepping into the node and pause at the next node.

While single-stepping, you will notice the following behaviors:

- When you single-step through code, nodes are highlighted to indicate they are ready to execute.
- Gray lines appear in a loop or a diagram to indicate that the section of code has finished executing but the program is still running.



Viewing Wire Data from the Previous VI Execution

You can configure a VI to retain wire values on the diagram so that when you create a probe, the probe displays the most recent data that flowed through the wire at the last VI execution.

Retaining wire values can help you debug a VI when a diagram is complex and you need to view specific wire data after a VI finishes executing to determine where an error occurred.

1. Click the **Retain wire values** button on the document toolbar to start retaining all wire values on the diagram.
2. Run the VI at least once so that wire values are immediately available to any

probes you create.

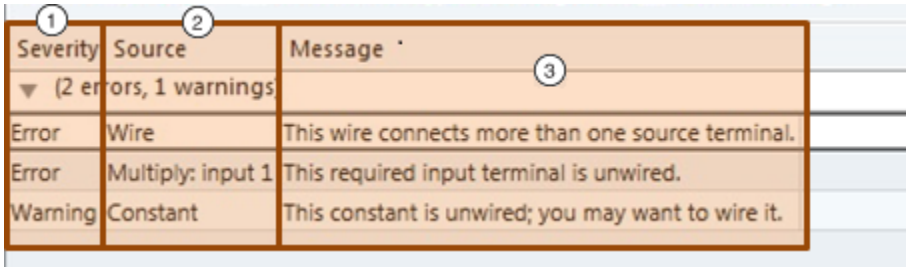
3. Place a probe on a specific wire by right-clicking the wire and selecting **Add probe**. To see the most recent value of the data that passed through the wire, either hover over the wire or click the probe.

Identifying Errors That Prevent You from Running Code

As you create code, a broken **Run** button communicates that the code contains errors that prevent it from running.

You can use the provided error and warning messages to help fix these problems. ***Errors*** break the code. You must resolve any errors before you can run the program. ***Warnings*** do not prevent you from running the code. They are designed to help you avoid potential problems in the program.

To identify the specific errors, click the broken **Run** button to display the Errors and Warnings tab. The following image highlights sections of the Errors and Warnings tab that define detected errors and warnings.



①	**Severity**—Denotes whether an issue is an error or a warning.
②	**Source**—Identifies the object that is causing the error or warning.
③	**Message**—Provides more detail about why the error or warning exists.

Improve Applications with Execution Logs

Well-written applications use execution logs for both activity audit and monitoring. An execution log makes it easy for a developer or user to track and identify issues that occur throughout the application without excessive effort.Maintaining an execution log allows you to review all activity and make improvements to your application accordingly.

An application should record data from multiple execution points throughout an application—errors, exceptions, successful execution, and so on. An execution log can help you recreate a certain behavior and improve your application effectively. For example, you can use the log to do one or more of the following:

- Modify the code to be able to handle a situation that caused an error
- Modify the code based on the most common use case
- Update the documentation for the application so that users interact with the application in a more predictable way
- Validate successful execution

The execution log should include as much information as you need to understand the state of the product at the time the data was collected. You might include one or more of the following components in each entry of an execution log:

- The current time and date
- A pre-defined category for the activity
- A running count of the specific type of activity
- A description of the activity
- The value of data that was generated during or prior to the activity
- The location of the activity within the application
- The type and severity of the problem, if there is one

You can use the String nodes to generate, combine, and write the text for each entry of an execution log to one or more text files. A log file should be human- and computer-readable, but you can format your file in a way that fits the needs of your application.

Viewing the Hierarchy of VIs in Your Application

Open the Call Hierarchy document to see the hierarchy of VIs and subVIs and understand the calling relationships between them in your application.

For example, you can use the Call Hierarchy document to see all of the VIs that call a specific VI and the subVIs that specific VI calls. This can be helpful when you need to debug a part of your application and want to see which subVIs are connected to each other.

1. Click **View** » **Call Hierarchy** to open a Call Hierarchy document for your project.

   Note Some callers may be hidden. Toggle the triangle icon below a node to show or hide callers.

2. (Optional) Click **Clean up diagram** to change the visual arrangement of the hierarchy.
3. (Optional) Select a subVI to highlight its calling connections.
4. (Optional) Double-click a subVI displayed in the Call Hierarchy document to open it.

Language Libraries

Multiple programming languages are available to accommodate your programming needs.

Language	Description
G Dataflow (G)	A graphical language in which data flows from left to right through wires and nodes.