Building the Block Diagram

After you build the front panel, you add code using graphical representations of functions to control the front panel objects. The block diagram contains this graphical source code.

**For more information...**
Refer to the LabVIEW Help for more information about designing and configuring the block diagram.

**Relationship between Front Panel Objects and Block Diagram Terminals**

Front panel objects appear as terminals on the block diagram. Double-click a block diagram terminal to highlight the corresponding control or indicator on the front panel.

Terminals are entry and exit ports that exchange information between the front panel and block diagram. Data you enter into the front panel controls enter the block diagram through the control terminals. During execution, the output data flow to the indicator terminals, where they exit the block diagram, reenter the front panel, and appear in front panel indicators.

**Block Diagram Objects**

Objects on the block diagram include terminals, nodes, and functions. You build block diagrams by connecting the objects with wires.

**Block Diagram Terminals**

You can configure front panel controls or indicators to appear as icon or data type terminals on the block diagram. By default, front panel objects appear as icon terminals. For example, a knob icon terminal, shown at left, represents a knob on the front panel. The DBL at the bottom of the terminal represents a data type of double-precision, floating-point numeric.
A DBL terminal, shown at left, represents a double-precision, floating-point numeric control or indicator. Right-click a terminal and select Display Icon from the shortcut menu to remove the checkmark and to display the data type for the terminal. Use icon terminals to display the types of front panel objects on the block diagram, in addition to the data types of the front panel objects. Use data type terminals to conserve space on the block diagram.

**Note** Icon terminals are larger than data type terminals, so you might unintentionally obscure other block diagram objects when you convert a data type terminal to an icon terminal.

A terminal is any point to which you can attach a wire, other than to another wire. LabVIEW has control and indicator terminals, node terminals, constants, and specialized terminals on structures, such as the input and output terminals on the Formula Node. You use wires to connect terminals and pass data to other terminals. Right-click a block diagram object and select Visible Items»Terminals from the shortcut menu to view the terminals. Right-click the object and select Visible Items»Terminals again to hide the terminals. This shortcut menu item is not available for expandable VIs and functions.

**Control and Indicator Data Types**

Table 5-1 shows the symbols for the different types of control and indicator terminals. The color and symbol of each terminal indicate the data type of the control or indicator. Control terminals have a thicker border than indicator terminals. Also, arrows appear on front panel terminals to indicate whether the terminal is a control or an indicator. An arrow appears on the right if the terminal is a control, and an arrow appears on the left if the terminal is an indicator.

<table>
<thead>
<tr>
<th>Control</th>
<th>Indicator</th>
<th>Data Type</th>
<th>Color</th>
<th>Default Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="SGL" /></td>
<td><img src="image" alt="SGL" /></td>
<td>Single-precision floating-point numeric</td>
<td>Orange</td>
<td>0.0</td>
</tr>
<tr>
<td><img src="image" alt="DBL" /></td>
<td><img src="image" alt="DBL" /></td>
<td>Double-precision floating-point numeric</td>
<td>Orange</td>
<td>0.0</td>
</tr>
<tr>
<td><img src="image" alt="EXT" /></td>
<td><img src="image" alt="EXT" /></td>
<td>Extended-precision floating-point numeric</td>
<td>Orange</td>
<td>0.0</td>
</tr>
<tr>
<td><img src="image" alt="CGL" /></td>
<td><img src="image" alt="CGL" /></td>
<td>Complex single-precision floating-point numeric</td>
<td>Orange</td>
<td>0.0 + i0.0</td>
</tr>
<tr>
<td>Control</td>
<td>Indicator</td>
<td>Data Type</td>
<td>Color</td>
<td>Default Values</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>-----------------------------------------------</td>
<td>--------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>Complex double-precision floating-point numeric</td>
<td>Orange</td>
<td>0.0 + i0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex extended-precision floating-point numeric</td>
<td>Orange</td>
<td>0.0 + i0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit signed integer numeric</td>
<td>Blue</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit signed integer numeric</td>
<td>Blue</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit signed integer numeric</td>
<td>Blue</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit unsigned integer numeric</td>
<td>Blue</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit unsigned integer numeric</td>
<td>Blue</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit unsigned integer numeric</td>
<td>Blue</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;64.64&gt;-bit time stamp</td>
<td>Brown</td>
<td>date and time (local)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enumerated type</td>
<td>Blue</td>
<td>—</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td>Green</td>
<td>FALSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>Pink</td>
<td>empty string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array—Encloses the data type of its elements in square brackets and takes the color of that data type.</td>
<td>Varies</td>
<td>—</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cluster—Encloses several data types. Cluster data types are brown if all elements of the cluster are numeric or pink if the elements of the cluster are different types.</td>
<td>Brown or Pink</td>
<td>—</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path</td>
<td>Aqua</td>
<td>&lt;Not A Path&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td>Blue</td>
<td>—</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>Indicator</td>
<td>Data Type</td>
<td>Color</td>
<td>Default Values</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>-----------</td>
<td>-------</td>
<td>----------------</td>
</tr>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
<td>Waveform—Cluster of elements that carries the data, start time, and Δt of a waveform. Refer to the <strong>Waveform Data Type</strong> section of Chapter 12, <em>Graphs and Charts</em>, for more information about the waveform data type.</td>
<td>Brown</td>
<td>—</td>
</tr>
<tr>
<td><img src="image3.png" alt="Image" /></td>
<td><img src="image4.png" alt="Image" /></td>
<td>Digital waveform</td>
<td>Dark Green</td>
<td>—</td>
</tr>
<tr>
<td><img src="image5.png" alt="Image" /></td>
<td><img src="image6.png" alt="Image" /></td>
<td>Digital data</td>
<td>Dark Green</td>
<td>—</td>
</tr>
<tr>
<td><img src="image7.png" alt="Image" /></td>
<td><img src="image8.png" alt="Image" /></td>
<td>Reference number (refnum)</td>
<td>Aqua</td>
<td>—</td>
</tr>
<tr>
<td><img src="image9.png" alt="Image" /></td>
<td><img src="image10.png" alt="Image" /></td>
<td>Variant—Stores the control or indicator name, information about the data type from which you converted, and the data itself. Refer to the <strong>Handling Variant Data</strong> section of this chapter for more information about the variant data type.</td>
<td>Purple</td>
<td>—</td>
</tr>
<tr>
<td><img src="image11.png" alt="Image" /></td>
<td><img src="image12.png" alt="Image" /></td>
<td>I/O name—Passes DAQ channel names, VISA resource names, and IVI logical names you configure to I/O VIs to communicate with an instrument or a DAQ device. Refer to the <strong>I/O Name Controls and Indicators</strong> section of Chapter 4, <em>Building the Front Panel</em>, for more information about the I/O name data type.</td>
<td>Purple</td>
<td>—</td>
</tr>
<tr>
<td><img src="image13.png" alt="Image" /></td>
<td><img src="image14.png" alt="Image" /></td>
<td>Picture—Displays pictures that can contain lines, circles, text, and other types of graphic shapes. Refer to the <strong>Using the Picture Indicator</strong> section of Chapter 13, <em>Graphics and Sound VIs</em>, for more information about the picture data type.</td>
<td>Blue</td>
<td>—</td>
</tr>
</tbody>
</table>

Many data types have a corresponding set of functions that can manipulate the data. Refer to the **Functions Overview** section of this chapter for information about which functions to use with each data type.

Refer to the **Memory and Speed Optimization** section of Chapter 6, *LabVIEW Style Guide*, in the *LabVIEW Development Guidelines* manual for more information about selecting a data type to optimize memory usage.
Constants

Constants are terminals on the block diagram that supply fixed data values to the block diagram. Universal constants are constants with fixed values, such as \( \pi \) (\( \pi \)) and infinity (\( \infty \)). User-defined constants are constants you define and edit before you run a VI.

Label a constant by right-clicking the constant and selecting Visible Items»Label from the shortcut menu. Universal constants have predetermined values for their labels that you can edit by using the Operating tool or the Labeling tool.

Most constants are located at the bottom or top of their palettes.

Universal Constants

Use universal constants for mathematical computations and formatting strings or paths. LabVIEW includes the following types of universal constants:

- **Universal numeric constants**—A set of high-precision and commonly used mathematical and physical values, such as the natural logarithm base (\( e \)) and the speed of light. The universal numeric constants are located on the Additional Numeric Constants palette.

- **Universal string constants**—A set of commonly used non-displayable string characters, such as line feed and carriage return. The universal string constants are located on the String palette.

- **Universal file constants**—A set of commonly used file path values, such as Not a Path, Not a Refnum, and Default Directory. The universal file constants are located on the File Constants palette.

User-Defined Constants

The Functions palette includes constants organized by type, such as Boolean, numeric, ring, enumerated type, color box, string, array, cluster, and path constants.

Create a user defined constant by right clicking an input terminal of a VI or function and selecting Create Constant from the shortcut menu. You cannot change the value of user-defined constants when the VI is running.

You also can create a constant by dragging a front panel control to the block diagram. LabVIEW creates a constant that contains the value of the front panel control at the time you dragged it to the block diagram. The front panel control remains on the front panel. Changing the value of the control does not affect the constant value and vice versa.
Use the Operating or Labeling tool to click the constant and edit its value. If automatic tool selection is enabled, double-click the constant to switch to the Labeling tool and edit the value.

**Block Diagram Nodes**

Nodes are objects on the block diagram that have inputs and/or outputs and perform operations when a VI runs. They are analogous to statements, operators, functions, and subroutines in text-based programming languages. LabVIEW includes the following types of nodes:

- **Functions**—Built-in execution elements, comparable to an operator, function, or statement. Refer to the *Functions Overview* section of this chapter for more information about the functions available in LabVIEW.

- **SubVIs**—VIs used on the block diagram of another VI, comparable to subroutines. Refer to the *SubVIs* section of Chapter 7, *Creating VIs and SubVIs*, for more information about using subVIs on the block diagram.

- **Structures**—Process control elements, such as Flat and Stacked Sequence structures, Case structures, For Loops, or While Loops. Refer to Chapter 8, *Loops and Structures*, for more information about using structures.

- **Formula Nodes**—Resizable structures for entering equations directly into a block diagram. Refer to the *Formula Nodes* section of Chapter 21, *Formulas and Equations*, for more information about using Formula Nodes.

- **Expression Nodes**—Structures for calculating expressions, or equations, that contain a single variable. Refer to the *Expression Nodes* section of Chapter 21, *Formulas and Equations*, for more information about using Expression Nodes.

- **Property Nodes**—Structures for setting or finding properties of a class. Refer to the *Property Nodes* section of Chapter 17, *Programmatically Controlling VIs*, for more information about using Property Nodes.

- **Invoke Nodes**—Structures for executing methods of a class. Refer to the *Invoke Nodes* section of Chapter 17, *Programmatically Controlling VIs*, for more information about using Invoke Nodes.

- **Code Interface Nodes (CINs)**—Structures for calling code from text-based programming languages. Refer to the *Code Interface Node* section of Chapter 20, *Calling Code from Text-Based Programming*.
Languages, for more information about calling code from text-based programming languages.

- **Call By Reference Nodes**—Structures for calling a dynamically loaded VI. Refer to the Call By Reference Nodes and Strictly Typed VI Refnums section of Chapter 17, Programmatically Controlling VIs, for more information about using Call By Reference Nodes.

- **Call Library Nodes**—Structures for calling most standard shared libraries or DLLs. Refer to the Call Library Function Node section of Chapter 20, Calling Code from Text-Based Programming Languages, for more information about using Call Library Nodes.

### Functions Overview

Functions are the essential operating elements of LabVIEW. Function icons on the Functions palette have pale yellow backgrounds and black foregrounds. Functions do not have front panels or block diagrams but do have connector panes. You cannot open nor edit a function.

The Functions palette also includes the VIs that ship with LabVIEW. Use these VIs as subVIs when you build data acquisition, instrument control, communication, and other VIs. Refer to the Using Built-in VIs and Functions section of Chapter 7, Creating VIs and SubVIs, for more information about using the built-in VIs.

### Numeric Functions

Use the Numeric functions to create and perform arithmetic, trigonometric, logarithmic, and complex mathematical operations on numbers and to convert numbers from one data type to another.

### Boolean Functions

Use the Boolean functions to perform logical operations on single Boolean values or arrays of Boolean values, such as the following tasks:

- Change a TRUE value to a FALSE value and vice versa.
- Determine which Boolean value to return if you receive two or more Boolean values.
- Convert a Boolean value to a number (either 1 or 0).
- Perform compound arithmetic on two or more Boolean values.
String Functions

Use the String functions to perform the following tasks:

- Concatenate two or more strings.
- Extract a subset of strings from a string.
- Search for and replace characters or subsets of strings in a string.
- Convert numeric data into strings.
- Format a string for use in a word processing or spreadsheet application.

Refer to the Strings section of Chapter 10, Grouping Data Using Strings, Arrays, and Clusters, for more information about using the String functions.

Array Functions

Use the Array functions to create and manipulate arrays, such as the following tasks:

- Extract individual data elements from an array.
- Add individual data elements to an array.
- Split arrays.

Refer to the Arrays section of Chapter 10, Grouping Data Using Strings, Arrays, and Clusters, for more information about using the Array functions.

Cluster Functions

Use the Cluster functions to create and manipulate clusters, such as the following tasks:

- Extract individual data elements from a cluster.
- Add individual data elements to a cluster.
- Break a cluster out into its individual data elements.

Refer to the Clusters section of Chapter 10, Grouping Data Using Strings, Arrays, and Clusters, for more information about using the Cluster functions.
Comparison Functions

Use the Comparison functions to compare Boolean values, strings, numerics, arrays, and clusters.

Refer to Appendix C, Comparison Functions, for more information about using the Comparison functions.

Time and Dialog Functions

Use the Time and Dialog functions to perform the following tasks:

- Manipulate the speed at which an operation executes.
- Retrieve time and date information from your computer clock.
- Create dialog boxes to prompt users with instructions.

The Time & Dialog palette also includes the Error Handler VIs. Refer to the Error Checking and Error Handling section of Chapter 6, Running and Debugging VIs, for more information about using the Error Handler VIs.

File I/O Functions

Use the File I/O functions to perform the following tasks:

- Open and close files.
- Read from and write to files.
- Create directories and files you specify in the path control.
- Retrieve directory information.
- Write strings, numbers, arrays, and clusters to files.

The File I/O palette also includes VIs that perform common file I/O tasks. Refer to Chapter 14, File I/O, for more information about using the File I/O VIs and functions.

Waveform Functions

Use the Waveform functions to perform the following tasks:

- Build waveforms that include the waveform values, channel information, and timing information.
- Extract individual data elements from a waveform.
- Edit individual data elements of a waveform.
Refer to Chapter 5, *Creating a Typical Measurement Application*, of the *LabVIEW Measurements Manual* for more information about creating and using waveforms in VIs.

**Application Control Functions**

Use the Application Control functions to programmatically control VIs and LabVIEW applications on your local computer or across a network. Refer to Chapter 17, *Programmatically Controlling VIs*, for more information about using the Application Control functions.

**Advanced Functions**

Use the Advanced functions to call code from libraries, such as dynamic link libraries (DLLs), to manipulate LabVIEW data for use in other applications, to create and manipulate Windows registry keys, and to call a section of code from text-based programming languages. Refer to the *Using External Code in LabVIEW* manual for more information about using the Advanced functions.

**Adding Terminals to Functions**

You can change the number of terminals for some functions. For example, to build an array with 10 elements, you must add 10 terminals to the Build Array function.

You can add terminals to expandable VIs and functions by using the Positioning tool to drag the top or bottom borders of the function up or down, respectively. You also can use the Positioning tool to remove terminals from expandable VIs and functions, but you cannot remove a terminal that is already wired. You must first delete the existing wire to remove the terminal.

You also can add or remove terminals by right-clicking one of the terminals of the function and selecting **Add Input**, **Add Output**, **Remove Input**, or **Remove Output** from the shortcut menu. Depending on the function, you can add terminals for inputs, outputs, or refnum controls. The **Add Input** and **Add Output** shortcut menu items add a terminal immediately after the terminal you right-clicked. The **Remove Input** and **Remove Output** shortcut menu items remove the terminal you right-clicked. If you use the shortcut menu items to remove a wired terminal, LabVIEW removes the terminal and disconnects the wire.
You transfer data among block diagram objects through wires. Each wire has a single data source, but you can wire it to many VIs and functions that read the data. You must wire all required block diagram terminals. Otherwise, the VI is broken and will not run. Display the Context Help window to see which terminals a block diagram node requires. The labels of required terminals appear bold in the Context Help window. Refer to the Setting Required, Recommended, and Optional Inputs and Outputs section of Chapter 7, Creating VIs and SubVIs, for more information about required terminals. Refer to the Correcting Broken VIs section of Chapter 6, Running and Debugging VIs, for more information about broken VIs.

Wires are different colors, styles, and thicknesses depending on their data types. A broken wire appears as a dashed black line with a red x in the middle. The arrows on either side of the red x indicate the direction of the data flow, and the color of the arrows indicate the data type of the data flowing through the wire. Refer to the LabVIEW Quick Reference Card for more information about wire data types.

Wire stubs are the truncated wires that appear next to unwired terminals when you move the Wiring tool over a VI or function node. They indicate the data type of each terminal. A tip strip also appears, listing the name of the terminal. After you wire a terminal, the wire stub for that terminal does not appear when you move the Wiring tool over its node.

A wire segment is a single horizontal or vertical piece of wire. A bend in a wire is where two segments join. The point at which two or more wire segments join is a junction. A wire branch contains all the wire segments from junction to junction, terminal to junction, or terminal to terminal if there are no junctions in between. Figure 5-1 shows a wire segment, bend, and junction.
While you are wiring a terminal, bend the wire at a 90 degree angle once by moving the cursor in either a vertical or horizontal direction. To bend a wire in multiple directions, click the mouse button to set the wire and then move the cursor in the new direction. You can repeatedly set the wire and move it in new directions.

To undo the last point where you set the wire, press the <Shift> key and click anywhere on the block diagram.

(Mac OS) Press the <Option> key and click. (UNIX and Linux) Click the middle mouse button.

When you cross wires, a small gap appears in the first wire you drew to indicate that the first wire is under the second wire.

⚠️ Caution Crossing wires can clutter a block diagram and make the block diagram difficult to debug.

Refer to the Wiring Techniques section of Chapter 6, LabVIEW Style Guide, in the LabVIEW Development Guidelines manual for more information about wiring tips and techniques.

**Automatically Wiring Objects**

LabVIEW automatically wires objects as you place them on the block diagram. You also can automatically wire objects already on the block diagram. LabVIEW connects the terminals that best match and leaves terminals that do not match unconnected.
As you move a selected object close to other objects on the block diagram, LabVIEW draws temporary wires to show you valid connections. When you release the mouse button to place the object on the block diagram, LabVIEW automatically connects the wires.

Toggle automatic wiring by pressing the space bar while you move an object using the Positioning tool.

By default, automatic wiring is enabled when you select an object from the Functions palette or when you copy an object already on the block diagram by pressing the <Ctrl> key and dragging the object. Automatic wiring is disabled by default when you use the Positioning tool to move an object already on the block diagram.

(Mac OS) Press the <Option> key. (Sun) Press the <Meta> key. (Linux) Press the <Alt> key.

You can disable automatic wiring by selecting Tools»Options and selecting Block Diagram from the top pull-down menu.

**Manually Wiring Objects**

Use the Wiring tool to manually connect the terminals on one block diagram node to the terminals on another block diagram node. The cursor point of the tool is the tip of the unwound wire spool. When you move the Wiring tool over a terminal, the terminal blinks. When you move the Wiring tool over a VI or function terminal, a tip strip also appears, listing the name of the terminal. If wiring to the terminal would create a broken wire, the cursor changes to a wire spool with a warning symbol. You can create the broken wire, but you must correct the broken wire before you can run the VI. Refer to the Correcting Broken Wires section of this chapter for more information about correcting broken wires.

Use the Context Help window to determine exactly where to connect wires. When you move the cursor over a VI or function, the Context Help window lists each terminal of the VI or function. The Context Help window does not display terminals for expandable VIs and functions, such as the Build Array function. Click the Show Optional Terminals and Full Path button in the Context Help window to display the optional terminals of the connector pane. Refer to the Setting Required, Recommended, and Optional Inputs and Outputs section of Chapter 7, Creating VIs and SubVIs, for more information about optional terminals.
Routing Wires

LabVIEW automatically finds a route for a wire as you wire it. LabVIEW routes a wire around existing objects on the block diagram, such as loops and structures. LabVIEW also routes a wire to decrease the number of bends in the wire. When possible, automatically routed wires from control terminals exit the right side of the terminal, and automatically routed wires to indicator terminals enter the left side of the terminal.

To automatically route an existing wire, right-click the wire and select **Clean Up Wire** from the shortcut menu.

Press the <Alt> key after you start a wire to temporarily disable automatic wire routing and route a wire manually. Press the <Alt> key again to enable automatic wire routing for the wire. After you end the wire, LabVIEW enables automatic wire routing again. You also can temporarily disable automatic routing after you click to start or set a wire by holding down the mouse button while you wire to another terminal or set point and then releasing the mouse button. After you release the mouse button, LabVIEW enables automatic wire routing again.

You can disable automatic wire routing for all new wires by selecting **Tools>Options** and selecting **Block Diagram** from the top pull-down menu.

If you disable automatic wire routing, you can wire terminals vertically or horizontally depending on the direction in which you first move the Wiring tool. The wire connects to the center of the terminal, regardless of where you click the terminal. After you click the terminal, press the spacebar to switch between the horizontal and vertical direction.

You also can press the spacebar to switch between the horizontal and vertical direction if automatic wire routing is enabled. If LabVIEW finds a route for the wire in the new direction, the wire switches to that direction.

Selecting Wires

Select wires by using the Positioning tool to single-click, double-click, or triple-click them. Single-clicking a wire selects one segment of the wire. Double-clicking a wire selects a branch. Triple-clicking a wire selects the entire wire.

Correcting Broken Wires

A broken wire appears as a dashed black line with a red x in the middle. Broken wires occur for a variety of reasons, such as when you try to wire
two objects with incompatible data types. Move the Wiring tool over a broken wire to display a tip strip that describes why the wire is broken. This information also appears in the Context Help window when you move the Wiring tool over a broken wire. Right-click the wire and select List Errors from the shortcut menu to display the Error list window. Click the Help button for more information about why the wire is broken.

Triple-click the wire with the Positioning tool and press the <Delete> key to remove a broken wire. You also can right-click the wire and select from shortcut menu options such as Delete Wire Branch, Create Wire Branch, Remove Loose Ends, Clean Up Wire, Change to Control, Change to Indicator, Enable Indexing at Source, and Disable Indexing at Source. These options change depending on the reason for the broken wire.

You can remove all broken wires by selecting Edit>Remove Broken Wires or by pressing the <Ctrl-B> keys. (Mac OS) Press the <Command-B> keys (UNIX) Press the <Meta-B> keys

⚠️ Caution Use caution when removing all broken wires. Sometimes a wire appears broken because you are not finished wiring the block diagram.

**Coercion Dots**

Coercion dots appear on block diagram nodes to alert you that you wired two different numeric data types together. The dot means that LabVIEW converted the value passed into the node to a different representation. For example, the Add function expects two double-precision floating-point inputs. If you change one of those inputs to an integer, a coercion dot appears on the Add function.

![Figure 5-2. Coercion Dot](image)

The block diagram places a coercion dot on the border of a terminal where the conversion takes place to indicate that automatic numeric conversion occurred. Because VIs and functions can have many terminals, a coercion dot can appear inside an icon if you wire through one terminal to another terminal.
Coercion dots also appear on the terminal when you wire any data type to a variant terminal, except when you wire two variants together. Refer to the *Handling Variant Data* section of this chapter for more information about the variant data type.

Coercion dots can cause a VI to use more memory and increase its run time. Try to keep data types consistent in your VIs.

**Polymorphic VIs and Functions**

Polymorphic VIs and functions can adjust to input data of different data types. Most LabVIEW structures are polymorphic, as are some VIs and functions.

**Polymorphic VIs**

Polymorphic VIs accept different data types for a single input or output terminal. A polymorphic VI is a collection of subVIs with the same connector pane patterns. Each subVI is an instance of the polymorphic VI.

For example, the Read Key VI is polymorphic. Its **default value** terminal accepts Boolean; double-precision, floating-point numeric; 32-bit signed integer numeric; path; string; or 32-bit unsigned integer numeric data.

For most polymorphic VIs, the data types you wire to the inputs of the polymorphic VI determine the instance to use. If the polymorphic VI does not contain a subVI compatible with that data type, a broken wire appears. If the data types you wire to the polymorphic VI inputs do not determine the instance to use, you must select the instance manually. If you manually select an instance of a polymorphic VI, the VI no longer behaves as a polymorphic VI because it accepts and returns only the data types of the instance you select.

To select the instance manually, right-click the polymorphic VI, select **Select Type** from the shortcut menu, and select the instance to use. You also can use the Operating tool to click the polymorphic VI selector, shown at left, and select an instance from the shortcut menu. Right-click the polymorphic VI on the block diagram and select **Visible Items** » **Polymorphic VI Selector** from the shortcut menu to display the selector. To change the polymorphic VI to accept all the handled data types again, right-click the polymorphic VI and select **Select Type** » **Automatic** from the shortcut menu or use the Operating tool to click the polymorphic VI selector and select **Automatic** from the shortcut menu.
Building Polymorphic VIs

Build polymorphic VIs when you perform the same operation on different data types.

**Note** You can build and edit polymorphic VIs only in the LabVIEW Professional Development System.

For example, if you want to perform the same mathematical operation on a single-precision floating-point numeric, an array of numerics, or a waveform, you could create three separate VIs—Compute Number, Compute Array, and Compute Waveform. When you need to perform the operation, you place one of these VIs on the block diagram, depending on the data type you use as an input.

Instead of manually placing a version of the VI on the block diagram, you can create and use a single polymorphic VI. The polymorphic Compute VI contains three instances of the VI, as shown in Figure 5-3.

![Diagram](image)

**Figure 5-3.** Example of a Polymorphic VI

The Compute VI statically links the correct instance of the VI based on the data type you wire to the Compute subVI on the block diagram, as shown in Figure 5-4.
Polymorphic VIs differ from most VIs in that they do not have a block diagram or a front panel.

Consider the following issues when you build polymorphic VIs:

- All VIs you include in the polymorphic VI must have the same connector pane pattern, because the connector pane of the polymorphic VI matches the connector pane of the VIs you use to create the polymorphic VI.
- The inputs and outputs on the connector pane of each instance of the VI must correspond to the inputs and outputs on the connector pane of the polymorphic VI.
- The VIs you use to build polymorphic VIs do not have to consist of the same subVIs and functions.
- Each of the front panels of the VIs do not have to have the same number of objects. However, each front panel must have at least the same number of controls and indicators that make up the connector pane of the polymorphic VI.
- You can create an icon for a polymorphic VI.
- You cannot use polymorphic VIs in other polymorphic VIs.

When you generate complete documentation for a VI that includes a polymorphic subVI, the polymorphic VI and its instances appear in the list of subVIs section of the documentation.