Originally released in 1986, LabVIEW (short for Laboratory Virtual Instrumentation Engineering Workbench) is a visual programming environment developed by National Instruments. LabVIEW is based on the “G” programming language which uses dataflow principles, rather than conventional procedural processing. Instead of executing one command at a time, dataflow programming languages consist of sets of instructions called nodes, connected by wires through which data flow. Since nodes execute immediately as data flows into them, concurrency (multiple threads of execution) can be implemented as simple as branching a wire, causing the nodes at both ends to execute at the same time. While LabVIEW is often only referred to as a programming language, it is also a complete integrated development environment (IDE) with a compiler, syntax checking, and debugging tools (Figure 2.1).

Figure 2.1: The LabVIEW “Getting Started” screen; the starting point for the LabVIEW IDE.
Many of the basic concepts of LabVIEW will be familiar to veteran programmers, and newcomers may find the interface to be more intuitive than in text-based programming languages. A simple example of a loop, for comparison, is shown in C and LabVIEW (Figure 2.2).

```
int x = 0;
for(int i = 1; i <= 10; i++)
{
    x += i;
}
```

*Figure 2.2: Summing numbers 1 to 10 in C-like languages (left), and the same calculations done in LabVIEW (right).*

In text-based languages, lines of code are executed in sequential order, with several structures which allow the order of execution to be modified. In this simple example, a variable, or a place to store data, is created and given the name “x”. The type of this variable is specified (an integer), and an initial value is given to it (0). Next, a “for” loop structure is encountered which executes a series of command multiple times. The header of this structure has 3 components divided by semi-colons: a variable initialization statement, a condition statement, and an update statement. The variable is created and given the value 1. The value held by this variable is checked to see if it meets the condition statement. If it does, all code between the following curly braces is executed (in this case, the value of i is added to the value of x). When this is done, the value of i is increased by one, it is again checked against the condition and the following block of code executes again. As long as the condition is met, this process of incrementing i and executing the code will continue.
In LabVIEW, the producing the same results looks considerably different. In place of the variable x, a numeric control named “Control” is used. LabVIEW has its own version of a “for” loop as well. It consists of a rectangular area in which code is placed, a square which connects to a value containing the number of time to loop (displayed as a square with the letter N), and an index value, similar to the variable i (it’s even represented by a square with the letter i). Unlike in the text language, the LabVIEW “for” loop doesn’t have an explicit condition, but rather always executes the code a number of times equal to the value connected to the N square. To reproduce the “add x and i, then store the result in x” logic, shift registers can be used in LabVIEW to obtain the value of “Control” from the previous execution of the loop, to which the index (i) is added. Since this index begins at 0, 1 must be added to being counting a 1 and end at 10.

Though LabVIEW can be used as a general purpose programming language, it lends itself most to data acquisition and control in that Application Programming Interfaces (APIs) for data transfer over numerous interfaces are provided. These API’s are generally quite extensive per interface, and can greatly simplify and shorten development.

A LabVIEW program consists of one or more virtual instruments (VIs) which are analogous to subprograms, such as functions or methods in other languages. Each VI consists of a front panel and a block diagram (Figure 2.3).
The front panel is the user interface of this VI, where the program’s interactive elements are placed. The front panel features two categories of interactive elements: controls and indicators. Controls represent input from either the user, or from other sections of code. Indicators, as the name suggests, represent output, whether to the user in the form of a text box or the value returned from a function. These structures may appear on the front panel, but they also act as containers for data, or variables, in the code. Not all VIs’ front panels are seen by the user (in general, most are not), but they can be a valuable debugging tool, since they can immediately display values from within the VI. The other component, the block diagram, is where all of the code is written.

The block diagram and front panel each provide a tool bar (Figures 4 and 5) consisting of buttons which represent many common debugging and coding operations.
The front panel’s tool bar buttons represent (from left to right):

- **Run** - Begins executing the current VI.
- **Run Continuously** - Executes the current VI, restarting if it terminates.
- **Abort Execution** - Cancels the current VI’s execution completely.
- **Pause** – Pauses the current VI’s execution, saving its current state to be restarted later.
- **Text Settings** – A menu containing options for text appearance, such as font, size, and justification.
- **Align Object** – A menu containing several methods for aligning the selected objects.
- **Distribute Objects** – A menu similar to Align Object with methods for evenly spacing the selected objects.
- **Resize Objects** – Similar to Align and Distribute objects; resizes front panel objects by one of several methods such that they are more uniform.
- **Reorder** – A menu allowing the selected object(s) to be grouped (such that all objects in a group are modified together), locked (preventing them from being modified), and their depth to be changed (which object will be on the top/bottom when they overlap).
- **Show Context Help Menu** – Opens a window which displays information about the object currently selected.

![Image of a tool bar menu]

**Figure 2.5: Block diagram tool bar and menu.**

The block diagram’s tool bar buttons represent (from left to right):

- **Run** - Begins executing the current VI.
• **Run Continuously** - Executes the current VI, restarting if it terminates.

• **Abort Execution** - Cancels the current VI’s execution completely.

• **Pause** – Pauses the current VI’s execution, saving its current state to be restarted later.

• **Highlight Execution** – When enabled, slows execution and displays the flow of data over wires.

• **Retain Wire Values** – When enabled, probes will retain their value until new data is passed through them.

• **Step Into (changes based on context)** – Progresses execution to the next step (such as a loop or subVI) when using Highlighted Execution.

• **Step Over (changes based on context)** – Progresses execution to the point where the next step (such as a loop for subVI) has terminated when using Highlighted Execution.

• **Step Out (changes based on context)** – Progresses execution until the current step (such as a loop or subVI) has terminated when using Highlighted Execution.

• **Text Settings** – A menu containing options for text appearance, such as font, size, and justification.

• **Align Object** – A menu containing several methods for aligning the selected objects.

• **Distribute Objects** – A menu similar to Align Object with methods for evenly spacing the selected objects.

• **Resize Objects** – Similar to Align and Distribute objects; resizes front panel objects by one of several methods such that they are more uniform.

• **Reorder** – A menu allowing the selected object(s) to be grouped (such that all objects in a ground are modified together), locked (preventing them from being modified), and their depth to be changed (which object will be on the top/bottom when they overlap).
• **Clean Up Diagram** – Automatically repositions the objects on the block diagram in an orderly manner, preferring readability over compactness.

• **Show Context Help Menu** – Opens a window which displays information about the object currently selected.

![Figure 2.6: LabVIEW functions palette.](image)

The code is created by arranging and connecting various structures and functions. All of these structures and functions usable in the program are located in the menu called the Functions Palette (Figure 2.6) from which they can be selected and placed on the block diagram. The Functions Palette is quite large, and until you’ve familiarized yourself with the location of the
most frequently used functions, the “Search” button located at the top of the palette is very helpful to find the program elements you need.

2.1 Basic Structures

The flow of data across wires and nodes is the most important concept when working with LabVIEW. The traditional “Hello World” program can help demonstrate this (Figure 2.7).

From the Functions Palette, expand the “Programming” section and click the “String” button. This displays a wide variety of functions for manipulating strings. Find the “String Constant” item, click it, and click again on the block diagram to place it. This is where we’ll store the text for our program. Settings its value is quite simple as well: simply double click on the item and a blinking cursor should appear. Now just type “Hello World” (quotation marks aren’t needed) and press enter or click outside of the item. Now switch to the front panel and open the Controls Palette. We need something of a user interface to see the text, so expand the “Modern” section, and open the “String & Path” sub-menu. We only want to display the text, so select “String
Indicator” and place it on the front panel. Going back to the block diagram, you should notice that the String Indicator is visible here too. You may have seen a purple dot on the edge of each item when holding your mouse cursor over it; these are the connectors; where you’ll be attaching wires to and from. If you hover your cursor over one the String Constant’s dot, you’ll notice that your cursor becomes a spool of thread. Click once and move the cursor to the dot of the String Indicator, and it snaps into place, showing a preview of the wire that will be made; click once again to actually create the wire. With that, the “Hello World” program is finished. Switch to the front panel and click the “Run” button (a white arrow pointing right) to see it in action. The data from the string constant flows over the wire and into the string indicator, which displays the value.

You may be wondering what dictates the direction of the data flow. The answer is that there are both input and output connectors. You may remember that the connector of the string constant was on the right side of the item. This means that data flows out of the item and into the connected wire. Conversely, the connector on the indicator was placed to the left, meaning that this connector accepts input. Basic indicators always have exactly one input, and basic constants and controls have exactly one output. Most functions follow the left-input, right-output convention (as seen in Figure 2.8), but it is not a requirement for user-created functions.

![Figure 2.8: The basic flow of data in LabVIEW.](image)
With the data flow concept in mind, data manipulation can be discussed. The constants, indicators, and controls described before are essentially only data containers. Operations on the data are done entirely with functions either provided by LabVIEW or created by users. As an example, two numeric constants are placed on the block diagram. In the same sub-menu, the Add function is placed on the block panel. Hovering the cursor over the Add function reveals three connectors: two on the left and one on the right. This function takes two numerical inputs and produces one output with the sum of the values. Double clicking the constants allows their values to be entered manually, and each constant is wired to a connector on the Add function. A third numeric constant is placed on the block diagram. To display the result of the addition, however, an indicator, rather than a constant, is needed. The new numeric constant can be changed to an indicator by right clicking it and selecting Convert to Indicator. In fact, nearly all constants, indicators, and controls may be freely converted to either of the other two types from their right click menu. The output of Add is connected to the new indicator and the front panel is opened.

Running the program produces output of “30” in the indicator. Becoming familiar with LabVIEW’s functions is a daunting task, but is beyond the scope of this introduction. LabVIEW provides extensive documentation of all functions (and structures), accessible by right clicking a particular function and clicking “Help”. Functions used in this book are briefly described as they’re needed.

By default, most controls and indicators are represented as a square icon on the block panel. These can also be displayed as a data type terminal. This can be changed by toggling View As Icon off in the right click menu. Figure 2.9 shows the differences between a string control’s icon and data type representation on the block diagram. Because the data type representation is much
smaller, it can be used to save space on the block diagram and is used almost frequently in this book.

Figure 2.9: The block diagram representations of a string control.

### 2.2 Strings

One of the most common and (as many programmers would agree) tedious things to work with are strings. A string is a series of numbers, letters, and other characters such as punctuation marks, whose data represents only the glyphs to be displayed. This means that strings which contain numbers do not have the numeric value they appear to, but rather the character codes to display each number. Extracting, formatting and interpreting numeric characters amongst the letters, punctuation, and whitespace is the source of much of the tedium.

Figure 2.10: Demonstration of the "String Subset" function.
LabVIEW thankfully provides an extensive library of string manipulation functions (such as the \textit{String Subset} function demonstrated in Figure 2.10); several of the most common are described in Table 2.1. The following projects make extensive use of strings for both instrument communication as well as user communication, so some familiarity with these functions is recommended.

![Diagram](image_url)

Table 2.1: Commonly used string functions and their descriptions.

<table>
<thead>
<tr>
<th>Function Name and Description</th>
<th>Connector Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>String Length</strong></td>
<td>string can be a string, a cluster of strings, or an array of clusters of strings. length has the same structure as string.</td>
</tr>
<tr>
<td>[ \text{string} \rightarrow \text{length} ]</td>
<td>[ \text{string} ]</td>
</tr>
<tr>
<td><strong>String Subset</strong></td>
<td>string is the input string. offset is the starting position and must be numeric. The offset of the first character in \text{string} is 0. If unwired or if less than 0, the default is 0. length must be numeric. If length is left unwired, the default is the length of string minus offset. substring is empty if offset is greater than the length of the \text{string} or if the length is less than or equal to 0. If length is greater than or equal to the length of string minus offset, substring is the remainder of string beginning at offset.</td>
</tr>
<tr>
<td>[ \text{string} \rightarrow \text{offset (0)} \rightarrow \text{length (rest)} \rightarrow \text{substring} ]</td>
<td>[ \text{substring} ]</td>
</tr>
<tr>
<td><strong>Replace Substring</strong></td>
<td>string is the string in which you want to replace characters. substring contains the substring that replaces length characters at offset in string. offset determines the number of characters into string at which the function places substring. length determines the number of characters in string to replace with substring. If string is empty, length characters are deleted starting at offset. result string contains the edited string with the replaced characters.</td>
</tr>
<tr>
<td>[ \text{string} \rightarrow \text{substring (&quot;&quot;} \rightarrow \text{offset (0)} \rightarrow \text{length (len. of substring)} \rightarrow \text{result string} \rightarrow \text{replaced substring} ]</td>
<td>[ \text{string}, \text{substring}, \text{offset}, \text{length} ]</td>
</tr>
</tbody>
</table>
The search for a regular expression in the input string beginning at the offset you enter and, if it finds a match, splits the string into three substrings and any number of submatches. Resize the function to view any submatches found in the string.

For components utilizing the PCRE library package, the following copyright notice applies.

Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyright by the University of Cambridge, England.

<table>
<thead>
<tr>
<th>Match Regular Expression</th>
<th>replaced substring contains the characters that were replaced in string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiline? (F)</td>
<td>multiline? sets whether or not to treat the text in input string as a multiple-line string. This affects how the ^ and $ characters handle matches. When FALSE (the default), entering &quot;^&quot; matches the beginning of the line in input string only and entering &quot;$&quot; matches the end of input string only. When TRUE, &quot;^&quot; matches the beginning of any line in input string and &quot;$&quot; matches the end of any line in input string.</td>
</tr>
<tr>
<td>ignore case? (F)</td>
<td>ignore case? specifies whether the string search is case sensitive. If FALSE (default), the string search is case sensitive.</td>
</tr>
<tr>
<td>input string</td>
<td>input string specifies the input string the function searches. This string cannot contain null characters.</td>
</tr>
<tr>
<td>regular expression</td>
<td>regular expression specifies the pattern you want to search for in input string.</td>
</tr>
<tr>
<td>offset</td>
<td>offset determines the number of characters into input string at which the function starts searching for search string.</td>
</tr>
<tr>
<td></td>
<td>before match returns a string containing all the characters before the match.</td>
</tr>
<tr>
<td></td>
<td>whole match contains all the characters that match the expression entered in regular expression. Any substring matches the function finds appear in the submatch outputs.</td>
</tr>
<tr>
<td></td>
<td>after match contains all characters following the matched pattern entered in regular expression.</td>
</tr>
<tr>
<td></td>
<td>offset past match returns the index in input string of the first character after the last match. If the VI does not find a match, offset past match is −1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>String</th>
<th>Match Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>string is the input string to search.</td>
</tr>
<tr>
<td>regular expression</td>
<td>regular expression is the pattern for which you want to search in string. If the function does not find regular expression, match substring is empty, before substring is the entire string, after substring is empty, and offset past match is −1.</td>
</tr>
<tr>
<td>offset (0)</td>
<td>You can use special characters to refine the search.</td>
</tr>
<tr>
<td>substring</td>
<td>offset is the starting position and must be numeric. The offset of the first character in string is 0. If unwired or if less than 0, the default is 0.</td>
</tr>
<tr>
<td>match substring</td>
<td>before substring returns a string containing all the characters before the match.</td>
</tr>
<tr>
<td></td>
<td>match substring is the matched string.</td>
</tr>
</tbody>
</table>

Searches for regular expression in string beginning at offset, and if it finds a match, splits string into three substrings. A regular expression requires a specific combination of characters for pattern matching. For more information about special characters in regular expressions, refer to the regular expression input description in the detailed help.
### after substring contains all characters following the matched pattern.

**offset past match** is the index in **string** of the first character of **after substring.** If the function does not find a match, **offset past match** is −1. The **offset** input and the **offset past match** output might be equal when the empty string is a valid match for the **regular expression.** For example, if **regular expression** is b* and the **string** input is cdb, **offset past match** is 0. If **string** is bbbcd, **offset past match** is 3.

### Format Into String

<table>
<thead>
<tr>
<th>format string</th>
<th>resulting string</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial string</td>
<td>error in (no error)</td>
</tr>
<tr>
<td>input 1 (0)</td>
<td>error out</td>
</tr>
<tr>
<td>input n (0)</td>
<td></td>
</tr>
</tbody>
</table>

Formats string path, enumerated type, time stamp, Boolean, or numeric data as text.

**format string** specifies how to convert the input arguments into **resulting string.** Defaults match the data type of the input arguments. Formatting a time stamp as anything other than time returns an error. Right-click the function and select **Edit Format String** from the shortcut menu to create and edit the format string. This input accepts a maximum of 255 characters.

**initial string** and any arguments appended to it form the **resulting string.**

**input 1..n** specifies the input parameters to be converted. Each input can be a string path, enumerated type, time stamp, or any numeric data type. You cannot use arrays and clusters with this function.

**resulting string** contains the concatenation of **initial string** and the formatted output.

### Concatenate String

<table>
<thead>
<tr>
<th>string 0</th>
<th>string 1</th>
<th>...</th>
<th>concatenated string</th>
</tr>
</thead>
</table>

Concatenates input strings and 1D arrays of strings into a single output string. For array inputs, this function concatenates each element of the array.

**string 0..n-1** are the strings you want to concatenate.

**concatenated string** contains the concatenated input strings in the order you wire them to the node from top to bottom.

Add inputs to the function by right-clicking an input and selecting **Add Input** from the shortcut menu or by resizing the function.
2.3  Wires

Wires are the primary medium through which data flows, and the primary means of controlling the flow of execution. As data reaches the end of a wire, it comes to a function or executable structure, and once all required data has reached that function or structure, it begins execution. An important property of wires is that they may carry only one type of data at a time. This means that the connectors to which the wire is attached must be of the same data type. When this is the case, the wire automatically changes to match that type. You may have noticed that wires can vary in color and thickness; this is to distinguish the data type flowing through the wire. An integer, for instance, creates blue wires that are a single pixel wide, and floating point numbers (that is, numbers which include a decimal point) wires are colored orange with a one pixel width.

Since one value is very frequently needed in more than one location, branching is an essential part of wires. Creating a wire branch (Figure 2.11) is as simple as clicking along the length of an existing wire (when the cursor is a spool of thread), and then clicking its end point. When data flows to a branching point, it is essentially copied, each flowing through a different branch simultaneously.
Very often when creating new wires, overlaps and unintentional segments are created. These can easily cause confusion to both yourself and others reading your code. “Cleaning up” these wires, however, is very simple; by right clicking on the offending wire and selecting *Clean Up Wire* and it is rearranged to the most straightforward path with as little overlapping as possible.

Similarly, broken wires (that is, wires without endpoints or have endpoints with non-matching data types) can be easily removed by right clicking the wire and selecting *Delete Wire Branch*, or, even more easily, simply press the Control key and “B” to delete all broken wires on the block diagram.

### 2.4 Conditional Processing

As with procedural text-based languages, LabVIEW makes extensive use of conditional statements. These case structures are displayed as rectangular blocks of codes with an outline based on the type of structure used (Figure 2.12).

![Figure 2.12: A case structure with important items labeled.](image)
In order to provide or extract data to/from code within these blocks, a “tunnel” must be created along the outline (Figure 2.13). A tunnel is simply an entry point for wires into structures which enclose other objects such as case structures.

![Figure 2.13: A numeric tunnel (the small blue square on the edge of the case structure outline) allows the value "0" to be accessed within the case structure.](image)

These tunnels appear as small squares and provide one input and one output to do nothing more than pass the data along; there’s no change in the data. Creating a wire across such a structure automatically creates a tunnel with the connections already made. Input to these structures is rather straightforward; data flows into it as it beings executing. Output, however, needs a bit more explanation. When outputting data from a case structure, there must be data connected to the tunnel for each case. If no such output is expected for certain cases, the tunnel can be right clicked to select “Use default if unwired”, which outputs the default value of that data type if the tunnel is unwired for the executed case.

We’ll begin with the case structure which is located in the Structures sub-menu in the Functions Palette. It acts most similarly to “switch” statements in C-like languages; it takes one input, and executes one of several “cases”, which contains code to execute when the input is equal to a certain value. Case structures appear as a rectangular area on the block diagram in which code
may be placed. When placing a case structure, two things of interest become apparent: the case name list at the top, and the case selector (input) on the left. A case structure is most similar to a switch structure, in that it contains blocks of code for each possibility. Clicking on the case name from the menu (or from the left/right arrows) displays the different blocks of code. The types of cases allowed depend on the type of input given (Figure 2.14). For example, a Boolean input has only True and False cases, while a numeric input can have a case of any numeric value within the range of the input type (such as 32-bit integers or floating point numbers). Managing the cases can be done from the right click menu of the case name list.

![Figure 2.14: Several possible numeric cases, with 0 as the default.](image)

Suppose a case structure is coded to handle integer inputs of 1, 2, and 3. If an integer value is wired to the case selector, the structure checks its value. If the value is 1, it executes the code given in the “1” case; if the value is 2, case “2” is executed, and so on. What if a value is given for which there isn’t a case? This is handled by the default case which executes when the input is equal to its case name or when the input doesn’t match any case. A case must be designated as the default, meaning it must be created with a valid case name first. Setting the default case is as simple as right clicking the case name and selecting “Make This Case Default”.
2.5 Looping

Loops are structures which continually execute code until a condition is met. Looping in LabVIEW is done primarily with *For* and *While* loops (Figure 2.15). These structures, similar to case structures, are represented by a rectangular area with distinctive borders in which tunnels may be created. Tunnels operate slightly different in loops, however. Data flowing into a loop are stopped at the tunnel and, in every iteration, the same data will be used in the loop’s code until the loop terminates. Tunneling data out of a loop, instead of outputting with each iteration, data flows out only after the loop has terminated.

![Figure 2.15: For loop (left), and While loop (right) with nodes labelled.](image)

“For” loops execute their area of code a specific number of times, incrementing a counter every iteration. The “For” loop contains two items of interest: an input for the number of iterations, and an output for the iteration counter. The number of iterations input must be provided with an integer value. Inside the loop, the output provides the code with the number of iterations that have executed thus far.
“While” and “For” loops are fundamentally the same, with their primary difference being the way they terminate. “While” loops, by contrast, do not execute a set number of times, instead they loop until a Boolean “True” enters the loop condition node (found by default at the lower right corner of the loop). Due to this specific stopping condition, great care needs to be taken when determining whether the loop should stop. Infinite loops or loops that never execute more than once are common errors which require re-evaluating the code’s logic.

“While” loops can be changed to terminate with a “False” value by right clicking the termination node and selecting “Continue When True”. It may, however, cause some confusion concerning your logic. If a loop would need to terminate in this way, the Boolean “Not” function applied to your input can be used to get the same effect while still being set to stop on a “True” value.

### 2.6 Sub VIs

Sub VIs are simply other VIs which are called from another VI- basically a function or method in more text-based languages. There is effectively no distinction from a VI and Sub VI, except that a Sub VI must provide input and/or output connectors. Creating a new Sub VI is as simple as opening the File menu and selecting “New VI”. Alternatively, you can click and drag a selection box around code on the current block diagram, and click “Edit”, then “Create Sub VI” to convert the selected code into an unnamed VI.

Data can be passed to these functions using controls and indicators within the VI. Indicators represent input to the VI; it could be said that the passed data is indicated to the VI. Conversely,
controls are used for output from the VI; they control the data leaving the VI. For example, if a VI needs a numeric value, numeric indicator is needed, which is wired to the rest of the code as needed. In order for LabVIEW to know which indicators and controls should have a connector when placed on the block diagram, they must be mapped to the connector pattern. A connector pattern is a division of a VI’s icon into rectangular regions (called terminals) which represent locations where wires may connect. These terminals may be assigned to a specific control or indicator within the VI, which links any wires connected at that terminal to the control or indicator when the VI executes.

![Figure 2.16: A simple Sub VI with inputs and output to be mapped.](image)

On the front panel, the icon in the upper right corner what the node for this VI looks like on the block diagram. Right clicking this icon displays a menu. From this menu, clicking Show Connections (Figure 2.16) displays the connection pattern in the icon’s place (Figure 2.17). Each square in this pattern represents a possible location for an input or output connector. If the VI needs more connectors than is shown, right click again, expand the “Show Connection Patterns” options, and select a pattern with enough connector locations.
Now, each input and output required by the VI must be mapped. This is done by clicking one of the squares in the connector pattern icon, then clicking the indicator or control whose data come into or go out of the connector in that location. Click once more in a blank space on the front panel to confirm each connector. Once these are mapped, the VI is ready to use.
Placing the node for the new VI is done with the “Select VI” option of the functions palette. This will let you browse your hard disk for the desired “.vi” file, and allow it to be placed on the block diagram.

In Figure 2.17, a terminal on connector pattern was clicked, followed by a click on the string control. This causes any string data wired to the (now pink colored) terminal when this VI is placed on a block diagram to be passed to the string control within the VI, allowing it to perform operations on it. Figure 2.18 shows the connector pattern once all controls and indicators for the VI have been mapped, as well as what the VI may look like when placed on a block diagram with wires connected. Figure 2.19 displays the block panel representation with the connector pattern displayed. This display of a VI can be done with any placed on the block diagram by right clicking the VI, selecting Visible Item, and then toggling Terminals.

2.7 Arrays and Clusters

Arrays are extremely powerful data structures, and are essentially ubiquitous. Arrays as they appear in LabVIEW function almost identically to their text-based counterparts. They store multiple values (of the same data type) accessible from within a single data structure. Each value
within the array occupies a sequential space in the computer’s memory. From the beginning of this sequential block of memory, each value can be determined using the size of each piece of data. This allows the values to be accessed via a numeric index corresponding to its location relative to the beginning of the array.

An important extension to the concept of arrays is nested or multi-dimensional arrays. This refers to arrays of arrays, in which each element of one array refers to its own independent array. Visually, if a one dimensional array is a series of values placed back-to-back, then a two dimensional array is a grid, and a three dimensional array is a cube. Accessing data in these types of arrays is a simple extension from one dimensional arrays. For each level of nesting, an additional index is required to specify a single point of data. In more visual terms, each dimension requires an index; for instance in a 2-D array, a width and height value are needed to specify a location, while in a 3-D array, width, height, and depth are needed.

As with everything else in LabVIEW, all operations are preformed with various functions. The “Array” menu of the functions palette provides a wide variety of array operations which include adding, changing, or reading a value at an arbitrary location within the array.

Clusters are somewhat similar to arrays with one major difference: clusters may contain data of multiple types. Due to this difference, however, a numeric index cannot be used to access data due to the varying sizes. Instead, values are accessed by the “Unbundle” function which separates the values, creating an output connector for each. In addition, the quantity and type of
values within the cluster must be known at compile time. This means values cannot be added or removed dynamically.

Perhaps the most common usage of clusters is in error handling. Many LabVIEW functions, particularly those involving I/O, have “Error In” and “Error Out” connectors. These pass along any error clusters until coming to an unwired node or an error handling function. When an error occurs within one of these functions, a cluster is assembled containing a Boolean flag, a 32-bit integer, and a string, which contain information about the error.

![Diagram](image)

*Figure 2.20: (Left) Accessing array element 2 with the “Index Element” function. (Right) Bundling and unbundling two values.*

### 2.8 Project Files

Common among the majority of IDEs, is the ability to organize files, compilation settings, and other useful data by storing their configurations in what is often called a project file. LabVIEW also uses project files for management of multi-VI programs. To create a new project, click “Empty Project” from the “Getting Started” window or “New Project” under the “Project” menu visible on any front panel or block diagram.

A view of the “Project Explorer” window is shown in Figure 2.21. From this window all associated files, build specifications, and more can be seen at a glance. Although individual VIs
may be created and call other VIs without an associated project, a project is required to create a standalone executable file. Though they provide a multitude of functionality, the most important is simply organization of all files associated with the project. Adding new files (or folders) can be done by right clicking the “My Computer” item in the project explorer and selecting “New”, then the desired file type for new files, or “Add”, then “File…” for pre-existing files. Files may also be added by simply dragging and dropping them onto the project explorer window.

![Project Explorer window](image)

Figure 2.21: The Project Explorer window.

### 2.9 Compilation and Execution

As with all programming languages, there must be a starting point for code execution. In LabVIEW, this starting point is the “Startup VI”. This is a similar concept as the “main” function in many other languages in which a function named specifically “main” is the origin of all other function calls in the program. Unlike many other languages, multiple starting points may be
selected. All of the selected “Startup VIs” will execute in parallel once the program begins. These VIs can be selected from within the properties menu of the desired build specification (Figure 2.22). All of the Startup VIs must be part of the same project as the build specification. A build specification is created by right clicking the “Build Specification” item in the Project Explorer window, selecting “New”, and choosing Application (EXE). The properties menu of the newly created specification is accessible from its right click menu. Creating the actual executable file is as simple as choosing “Build” from the right click menu of a build specification.

![My Application Properties](image)

Figure 2.22: The Build Specification properties window.

An interesting property of LabVIEW is that a VI can be executed manually, in its current state, from within the IDE without going through the process of building an executable. This is done by clicking the right-facing arrow button on the toolbar (the “Run” button). Before a VI can execute, it must be free from syntax errors. If any syntax errors exist within the current VI, the
“Run” button will change to a broken arrow icon. Clicking it at this time will display a list of all syntax errors. Once a VI is executing, it may be halted immediately with the red, octagonal “Stop” button, near the “Run” button. Execution can also be paused with the “Pause” button, represented by two vertical lines. During development, this individual VI execution is most common, as it allows for quick testing and usage of the IDE’s debugging capabilities.

2.10 Debugging

At the core of debugging in LabVIEW is a feature known as “Highlighted Execution”. On the block diagram, this feature drastically slows the VI’s execution for interpretation by the programmer, and creates a visual representation of data flowing through the wires of the program (Figure 2.23). It can be toggled on and off at any time (even while the VI is already running) by clicking the light bulb icon on the toolbar. Highlighted execution takes place in steps, each of which represents all pathways which may currently be executed. The Step Into, Step Over, and Step Out menu item located on the toolbar allow the programmer to jump to only the code to test.

Figure 2.23: Highlighted Execution displaying the values at each wire.