KU LEUVEN kulak

Faculteit Wetenschappen

Probleemoplossen en Ontwerpen

Introduction to LabVIEW

Graphical programming

Erik Verboven
Kevin Truyaert

KU Leuven Kulak, Wetenschap & Technologie

Academiejaar 2020-2021

Contents

List of Figures

Introduction

LabVIEW (Laboratory Virtual Instrument Engineering Workbench) is a graphical programming language distributed by National Instruments since 1986. The language is based on the concept of Graphical Structured Data Flow ("G" in short). The graphical component means programming is not based on lines of text code, but on placing function blocks in diagrams on a worksheet. The data flow component indicates the execution order is not determined by the place on the block diagram but on the flow of data that is visualized by connection wires linking blocks on the diagram. A block of code is executed from the moment all inputs have been provided.

An important advantage of this approach is the simultaneous execution of tasks on different available cores and threads that is handled automatically as opposed to more traditional languages that often require intensive rewriting of code.

Originally the software was developed for engineers and experimental scientists that had limited programming training but were still required to control complex instruments and automation processes. Today the LabVIEW software is no longer limited to this purpose and with the ability of different toolkits has applications going from advanced signal processing to OS control mechanisms and numeric computations.

History

LabVIEW version 1.0 was released in 1986 on the Apple Macintosh that was introduced a few years earlier. It was developed starting from the idea of translating the attraction of the Mac's graphical interface to a programming language capable of controlling several box-instruments and automation of systems.

Starting from 1992 and version 2.5 the software became available to Windows clients and version 5.0 released in 1999 introduced LabVIEW to Linux systems.

From LabVIEW 8.0 (2005) on the major versions are released in the first weeks of August.

Course Overview

This course will give a short introduction to the LabVIEW programming language. At the end of the course the student should be able to understand the basic concepts of Virtual Instruments, data flow, program execution, subVIs (chapter 1), common data types (chapter 2), basic structures (chapter 3), error

handling (chapter 4), design patterns (chapter 5), instrument control (chapter 6) and the project environment (chapter 7).

It is important to note that this course is written based on LabVIEW 2015. Although most concepts should be applicable to most versions it is possible that some functions changed compared to older versions.

Chapter 1

Basics

This chapter treats the basic concepts of the LabVIEW programming language. After reading through this chapter you should be able to create and execute a simple VI and use the debugging tools provided.

1.1 Virtual Instrument (VI)

The **Virtual Instrument** or **VI** is the main building block of LabVIEW. The name stems from the original function of these blocks as replacements of functionality of physical instruments. A VI typically contains instructions for a particular operation or function. This is also reflected in the structure of the VI that is divided into a front panel and a block diagram.

Upon starting LabVIEW, a new VI can be opened by clicking File→New VI or by using the short-key Ctrl+N. This will open two windows.

1.1.1 Front Panel

The **front panel** of a VI is the graphical user interface (GUI) of the VI. Here you can place inputs for your program, monitor outputs and give instructions. The empty default panel has a grey background and a grid to ease placement of components.

Right-clicking in the panel brings up a **controls palette** (figure 1.1). This palette is a tree structure of different components that can be placed on the panel. Components that require the user to input values are called **controls**, components that output data are called **indicators**. The full palette can be expanded by clicking the bottom downward pointing double arrow. The palette is organized into different styles (modern, classic, system, silver) and frameworks (f.e. Signal Processing, ActiveX, .NET). These parts are further divided into numerics, booleans, strings, arrays, lists, instrument I/O, decorations, etc. The palette can also be searched by using the search function in the right top corner of the palette. In order to avoid the palette to disappear after each drop of a component the palette can be pinned to the front panel by clicking the pin symbol in the left top corner of the palette.

When expanded the controls palette shows the option (bottom) to change the palettes that are visible when the panel is in its non-expanded form. Further

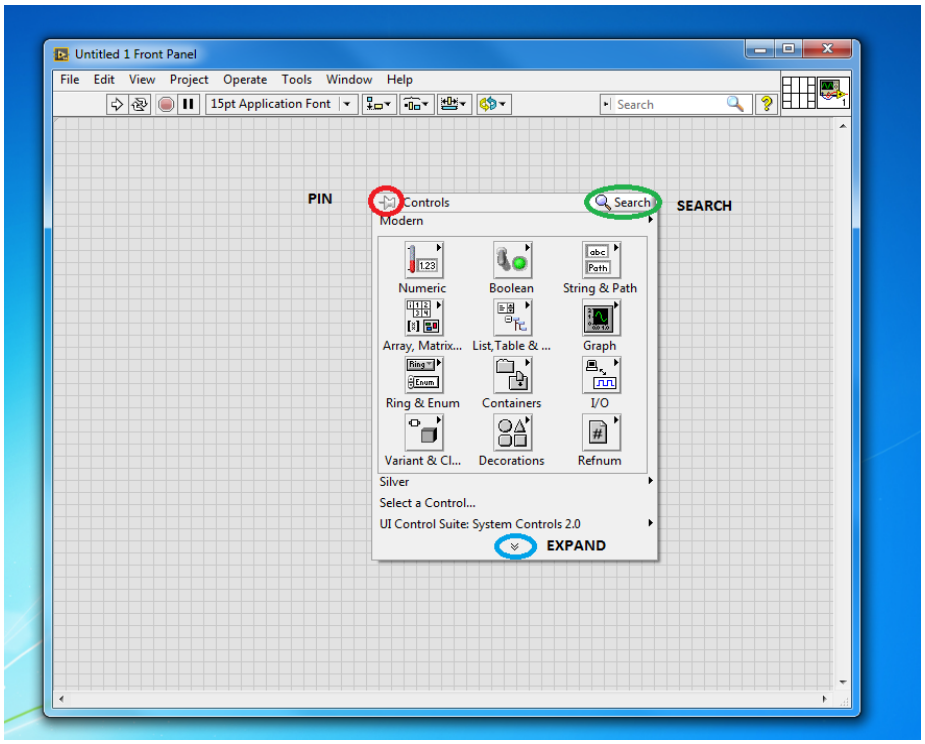

Figure 1.1: VI Front Panel with open Controls Palette (right-click panel). The palette can be pinned to the panel (red), searched (green) and expanded (blue).

customization options can be chosen when the palette is pinned to the panel.

A component can be placed on the panel simply by dragging and dropping the chosen control or indicator from the controls palette to the panel.

Browsing to the available controls and indicators in the controls palette it can be seen that there is a multitude of possibilities for representing data. For example a temperature readout can be represented by different numerics, gauges, meters, thermometers as can be seen in figure 1.2. This diversity allows to create attractive and accessible GUI for complex applications (figure 1.3). More about front panel customization techniques can be found in chapter 2.

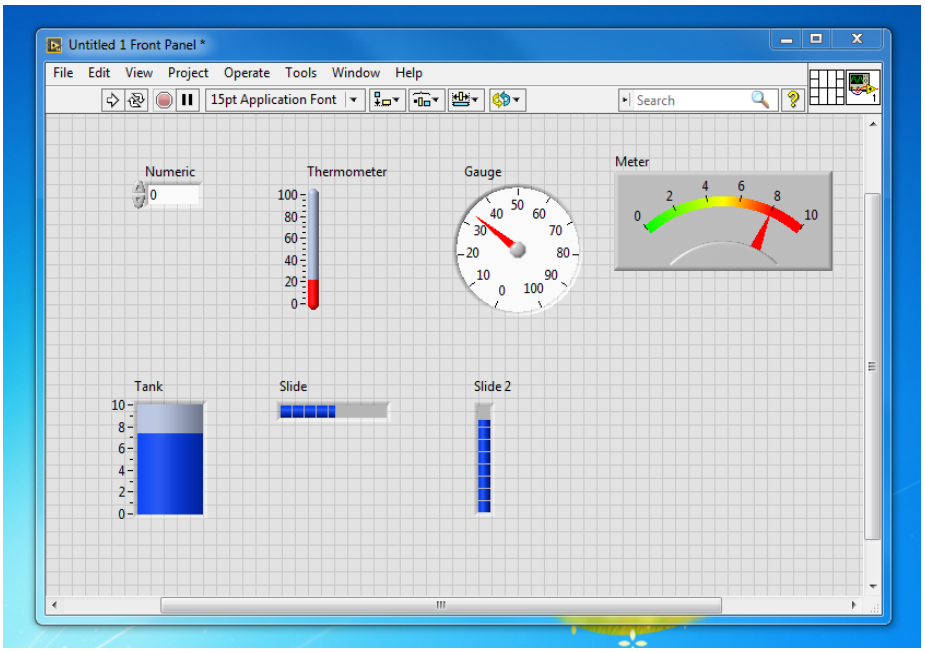

Figure 1.2: VI Front Panel with different style controls and indicators. These tools can aide in making a clear GUI for the user.

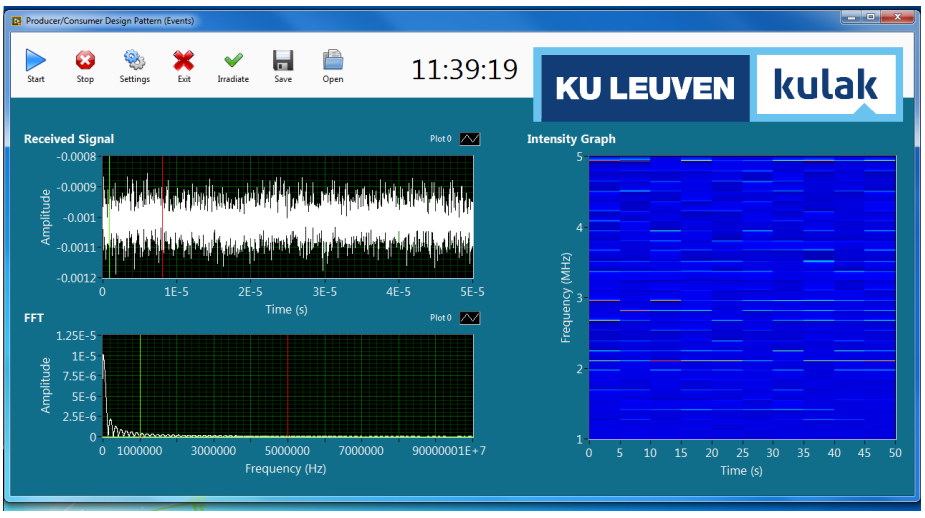

Figure 1.3: Example of a GUI in LabVIEW.

1.1.2 Block Diagram

The **block diagram** is the actual programming panel. Here function blocks will be placed linked by wires that indicate **data flow**. The diagram has a white

background and no grid by default.

By analogy with the front panel a **functions palette** can be brought up to select programming blocks (figure 1.4). This palette can likewise by pinned to the diagram, searched and expanded. The blocks can be dragged and dropped on the diagram. Take into account that it may take some time before the search function is ready loading.

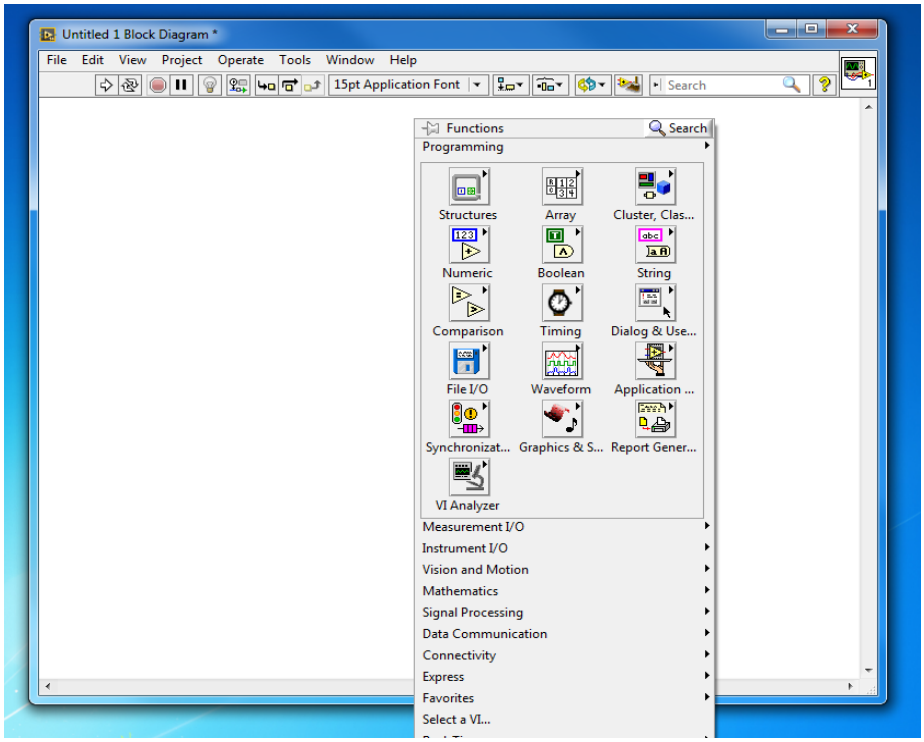

Figure 1.4: VI Block Diagram with open Functions Palette (right-click panel).

If a control or indicator is placed on the front panel a representation on the block diagram will also be available (figure 1.5). Double clicking the control on the front panel will make the block diagram pop up and show the location of the control on the diagram. Inversely double clicking the block diagram representation will pop up the front panel showing the control's location. On the block diagram you can also place **constants** for values that you don't want to show and/or change on the front panel (figure 1.6). The color of the block diagram representation of the control/indicator/constant depends on the type, for example numeric double (orange), numeric integer (blue), string (purple), boolean (green).

Once a function or control is placed on the diagram it can be connected by wires (figure 1.7). This can be accomplished by hovering the mouse over the control or indicator. A small dot will appear (on the lefthand side for an indicator and on the righthand side for a control). This dot is the start or endpoint of a wire. If you hover the mouse over the dot, the mouse icon will change to a spool to indicate you are ready to draw a wire. Clicking will start

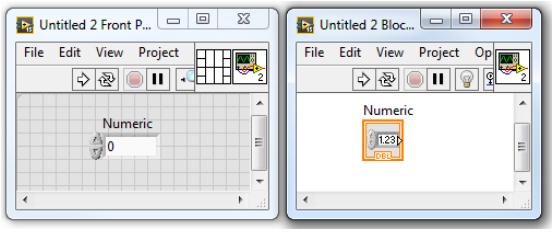

Figure 1.5: Numeric control displayed in the front panel (left) and the block diagram (right). The color of the block diagram representation depends on the type of control or indicator.

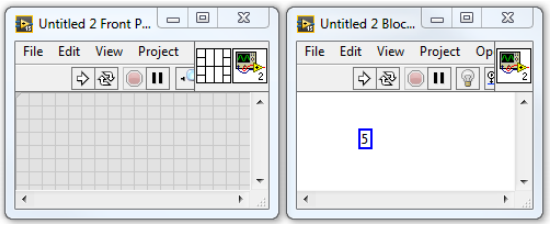

Figure 1.6: Integer constant displayed in the front panel (left) and the block diagram (right). As can be seen the constant does not appear on the front panel

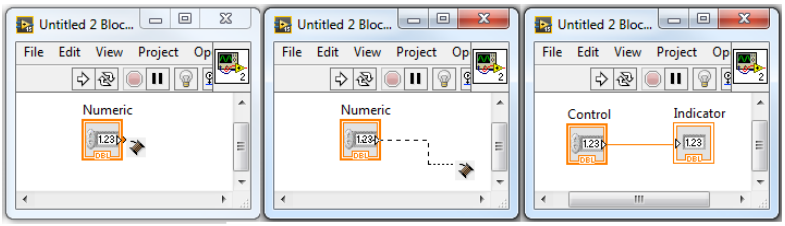

Figure 1.7: Wire drawing. a) Hovering over the input or output of a control/constant/indicator/function, b) Clicking the start point and dragging the wire, c) Once connected to an end point the wire has the color of the type of information that is passed.

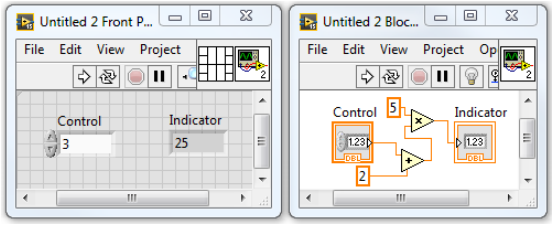

Figure 1.8: Although the multiplication is situated higher up than the addition, data flow rules dictate that the addition is calculated first.

the **wire drawing** mode. If you move the mouse in this mode, you will see a dotted wire starting from the clicked dot that follows you mouse movements.

A wire will be drawn once it is connected to another dot. Wire start points on the lefthand side of a function are inputs the function needs before it can execute, while wire start points on the righthand side of a function are outputs generated once the function is executed.

Connected controls, indicators, constants and functions will obey the rules of data flow, meaning a function block will be executed once all input wires deliver values to the function regardless of that function's location on the block diagram (figure 1.8).

1.1.3 Tools palette

The **Tools palette** is a palette that can be used on both the front panel and the block diagram (figure 1.9). It allows the user to select different operation modes of the mouse indicator. By selecting Views → Tools Palette the palette is opened. In normal circumstances the top button is selected. This is the automatic tool selection. When this mode is on, LabVIEW will automatically select the most like tool. If you for example hover near a function's input or output, LabVIEW will select the spool so that you can start drawing wires. If the mode is off, the user has the freedom to select the modes himself.

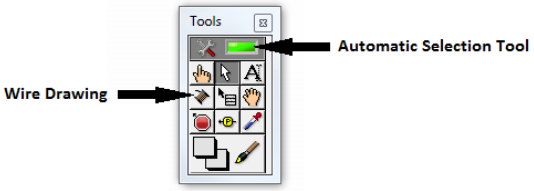

Figure 1.9: Tools Palette which allows to select different modes. If the automatic selection is on, LabVIEW will automatically change the mode.

1.2 Program Execution

A VI can be executed by clicking the white **Run Button** on the toolbar (figure 1.10). This will run the entire program once. During running the arrow will become black. If the arrow is gray and broken, it means the program cannot be run because of a programming error. Clicking the broken arrow will bring up the error list which provides more information on the problem (figure 1.11).

Running the program **continuously** requires pressing the looped arrows button.

In order to **abort** the execution of a VI before it has completely executed, the abort button needs to be pressed. **It should be noted that the abort button should only be used if the program got stuck unexpectedly or when an emergency stop is required**. In other cases the programmer is expected to implement a fitting shutdown procedure (chapter 3).

The execution can also be paused by means of the **pause button**. Clicking the button brings up the block diagram. Execution can be resumed by pressing the button again.

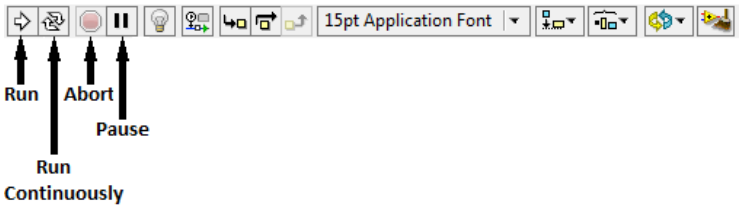

Figure 1.10: Execution tools. The run button will run the program once, run continuously will run the program in a loop. Execution can be stopped by pressing the abort button. This is for emergency stops only! The pause button halts execution and resumes it when pressed again.

Figure 1.11: Broken execution buttons. Pressing the button will bring up an error list with errors that need to be fixed before the program can run.

1.3 Debugging

During execution the block diagram can be monitored in several ways (figure 1.12). Pressing the light-bulb icon will **highlight the execution**, meaning that dots travel down the different wires indicating execution order. Furthermore function block outputs, controls and constants will indicate the values that are outputted. This feature will slow down execution considerably. Another method for debugging is the use of probes. Each wire of interest can be probed by right-clicking the wire and selecting **Probe**. A rectangle with a number will appear on the wire and a **Probe Watch Window** is opened. During execution the values passing on the probed wires including the time the last update occured will be given in the Watch Window.

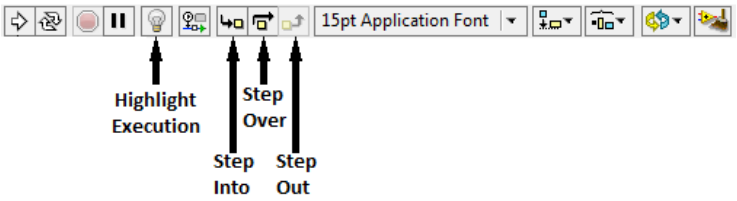

Figure 1.12: Debugging Tools. Highlight execution will slow down the program and allows to follow the actual data flow. The stepping tools allow to single step through the execution.

Another set of debugging tools are the **stepping tools**. These tools allow to step through the execution and pause at each node. If the node is of interest the user can choose to **Step Into** the module. To finish executing the node the user can choose to **Step Out**. If the user does not wish to debug a particular node he can choose to **Step Over** and pause at the next node.

1.4 SubVI

Good programming practice means you will have to group functionality in separate VIs so that you can reuse the code in other programs. Such a VI used in another main VI is called a **subVI**. It is important to notice that this subVI is a normal VI, so this is just a naming convention.

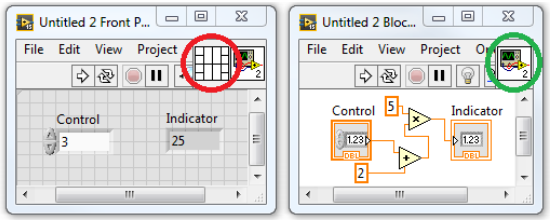

Figure 1.13: Location of the connector pane on the front panel window of a VI (red circle). Location of the VI icon (green circle) that shows the representation of the VI when used on the block diagram of another VI.

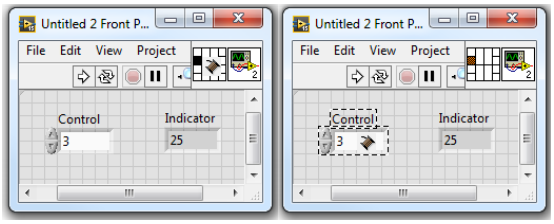

Figure 1.14: Connecting a control to the connector pane. Click an empty connector and then click the desired control. If successfully connected the connector slot will turn into the same color as the control on the block diagram.

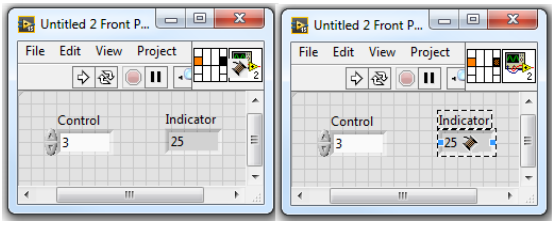

Figure 1.15: Connecting an indicator to the connector pane. Click an empty connector and then click the desired control. If successfully connected the connector slot will turn into the same color as the indicator on the block diagram.

In order for a VI to be usable as a subVI its relevant inputs and outputs need to be exposed to the main VI. To do this you can click an empty (white) space on the **connector pane** in the upper right hand corner of the front panel (figure 1.13). You can see the standard pattern has 4 terminals on the left, 4 on the right and two on the top and bottom. Other configurations can be chosen by right-clicking the pane and selecting patterns. It is however advised

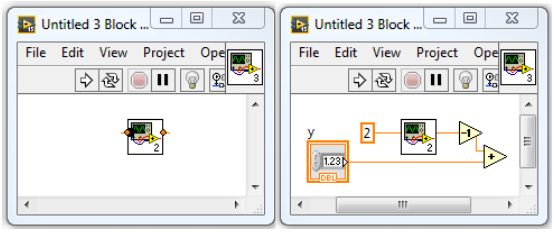

Figure 1.16: VI from figures 1.14 and 1.15 placed on the block diagram of another VI. Because the control and indicator were connected on the connector pane, they are exposed to the main VI's block diagram.

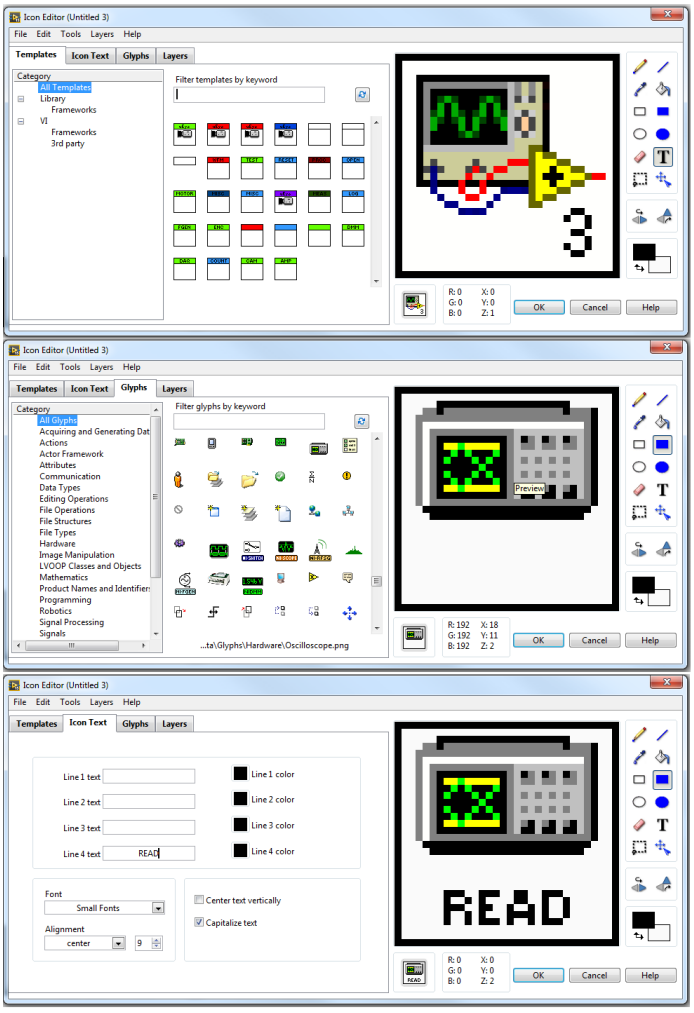

Figure 1.17: The Icon Editor allows the user to customize the look of a VI if used on the block diagram of another VI. Under the glyphs tab several symbols can be found. The icon text tab allows the user to put text in several lines.

to use the standard pattern as much as possible (this will be explained later on). When you selected an empty space it turns black and you can simply click the desired control or indicator that you want to link to this terminal (figures 1.14 and 1.15). If the terminal was successfully assigned it will be given the same color as the block diagram representation of the control or indicator. As with predefined functions the convention is to use the left part of the pane for inputs and the right part for outputs.

Any VI can be placed within another VI by choosing *Select a VI* (figure 1.16). The icon of the subVI on the block diagram is shown in the upper right corner of both the front panel and block diagram of the subVI. It can be changed by double clicking the icon. This will bring up the **Icon Editor** (figure 1.17). Simple paint tools, glyphs and text can be used to make a unique looking icon. This will help identify functionality on a block diagram.

1.5 Help features

LabVIEW offers several features that the user can exploit to get help. The first is the **context help** which can be activated by going to the menu bar and selecting Help → Show Context Help or pressing Ctrl+H. This feature shows a popup window whenever the user hovers over a function giving a short description of the function as well as the type of inputs and outputs that are expected. Furthermore the menu also has a link to the detailed help file.

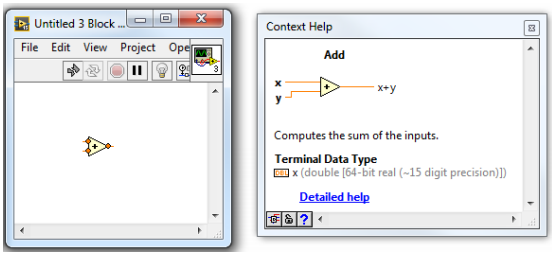

Figure 1.18: Context Help window popping up when hovering over the add function. It gives a short description of what the function does and the type of inputs and outputs that are provided.

The **Example Finder** is another great tool. It contains a vast number of example programs that come with explanations on the front panel and block diagram. The Example Finder can be opened from the menu bar Help → Find Examples. Once it is opened you have to choice of typing in specific search terms or browsing the different folders sorted by task. Take a special look at the Fundamentals folder.

The last help feature is the detailed **LabVIEW help system**. It can be opened from the menu bar Help → LabVIEW help. This help system **should be your best friend**. It has information about programming concepts, information about LabVIEW VIs, functions, palettes, menus and tools and step-by-step instructions.

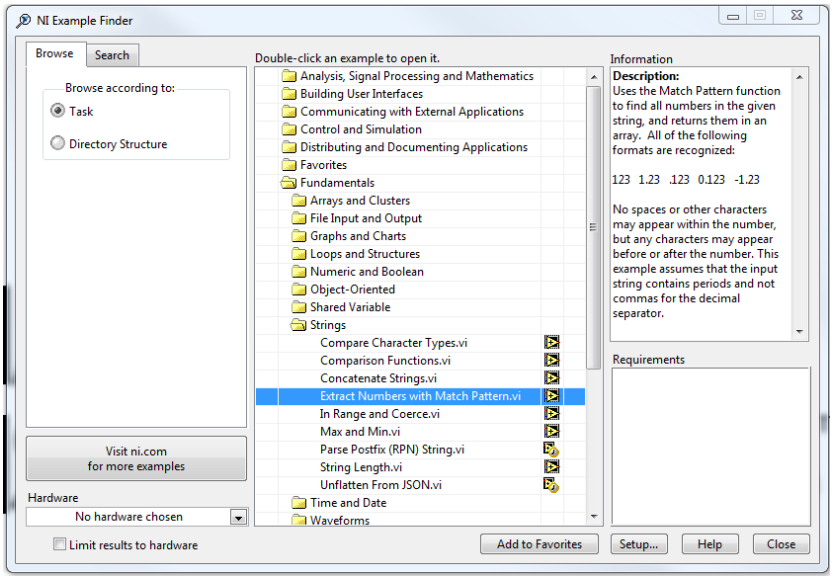

Figure 1.19: The Example Finder lets you browse or search a large database of well documented examples.

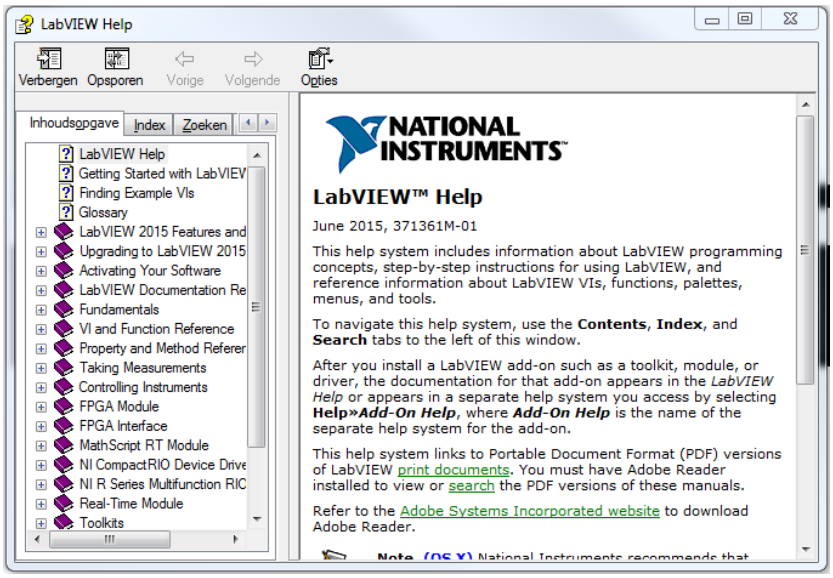

Figure 1.20: The LabVIEW help system is an indispensable tool when programming in LabVIEW.

1.6 Good programming practices

As in any language there are some simple practices to keep in mind when programming. Along the way this list will grow. For now you should keep to some simple rules.

- Before actually coding, always think about the problem ahead. Make a simple design of the way you want your front panel to look like and a work out a solution path.
- Keep your block diagram clean. This means avoiding wires crossing each other as much as possible, putting related pieces of code in subVIs, etc. An important rule of thumb: **keep your block diagram limited to the size of a computer screen**.
- Write comments in your code explaining the different steps. This is as easy as double-clicking the block diagram and start writing.

1.7 Exercises

1. Create a VI that is capable of converting a temperature value indicated in degrees Celsius to degrees Fahrenheit. The input and output values have to be visible on the Front Panel. You can play around with the available representation tools to make your front panel fancy. Hint: common numeric operations such as addition, multiplication etc, can be located in the functions palette under Programming → Numeric.
2. Create a VI that converts the value indicated in degrees Fahrenheit to degrees Celsius. The input and output values have to be visible on the Front Panel. You can play around with the available representation tools to make your front panel fancy.
3. Use the different debugging options (Highlight Execution, Probe, Stepping) to investigate the execution of your program.
4. Make your VI ready for use as a subVI in another program. Also use the icon editor to give your VI icon a fitting look.
5. Make a VI that gives the user the choice to enter a value in degrees Celsius or degrees Fahrenheit and convert it to degrees Fahrenheit or degrees Celsius respectively. Use the VIs that you already created for this purpose.

Chapter 2

Data types

This chapter explores the different data types that LabVIEW offers, the use of arrays of one particular type of data as well as clusters of different data types within the LabVIEW environment.

2.1 Basic types

2.1.1 Numeric

After going through chapter 1 you should already be familiar with the basic numeric data type: **double**. Another popular numeric data type is the **integer**. Both data types can be represented by the same front panel controls. On the block diagram the double and integer can be distinguished by their color (Double: orange, Integer: blue) (figure 2.1). The user can switch between these types by right-clicking the control/indicator and selecting *Representation*. Apart from Double precision and Long (32-bit integer) there are other numeric options that will not be discussed here.

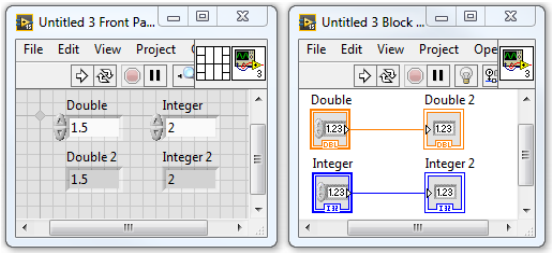

Figure 2.1: This VI shows two different numeric data types: double and integer. On the front panel the controls and indicators are similar. On the block diagram the difference is visualized by color (orange for double, blue for integer).

If a numeric of a certain type of numeric is connected to a function that expects a different numeric type, it will not show as an error, but a red dot, called a **coercion dot**, will appear on the connection point of the wire with the function (figure 2.2). LabVIEW will automatically convert to the data type of the destination. Be very careful with this, as this operation can lead to loss

of precision along the way. It is therefore recommended that you always use conversion functions (Programming → Numeric → Conversion) to first convert to the correct data type, so that you are fully aware of the conversion.

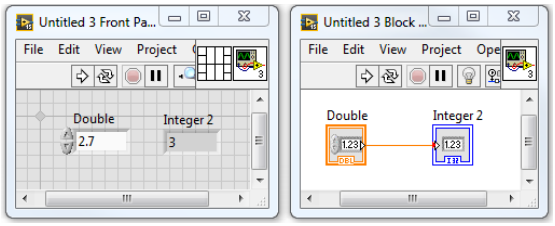

Figure 2.2: When numerics of a different type are connected the connection point of the wire will show a red dot indicating that the data will be converted to another data type.

The processes that can be used on numeric data can be found on the block diagram under Programming → Numeric.

2.1.2 Boolean

A very common data type in programming logic is the **boolean**. It can have only two values: true or false. This data type has its own set of front panel controls and indicators comprising switches, LEDs, push buttons, radio buttons, etc. It is represented on the block diagram in green. Boolean wires are green dotted.

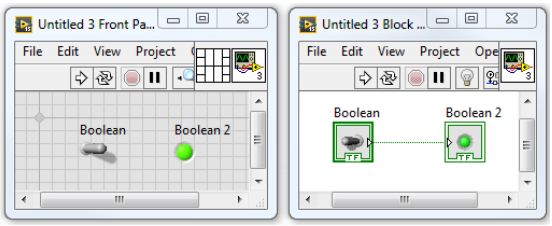

Figure 2.3: Boolean control and indicator example. The connecting wire is not solid as for numeric data, but dotted.

The operations that can be performed with booleans can be found on the block diagram under Programming → Boolean.

2.1.3 String

The **string** data type can contain a variable number of ASCII characters. Its block diagram color is pink/purple and string wires have a wavy pattern. Again this data type has its own front panel control and indicator. Operations that can be performed with strings can be found under Programming → String. For saving purposes it is often required to convert numerics to strings and vice versa. Therefore the string conversion tools are of particular interest. They can be found under Programming → String → Number/String conversion.

A data type closely related to the string data type is the **path** (figure 2.5). As the name suggest this data type is used for absolute and relative directory

paths. The front panel control and indicator can be found under the same division as the strings. Conversion tools to string are readily available.

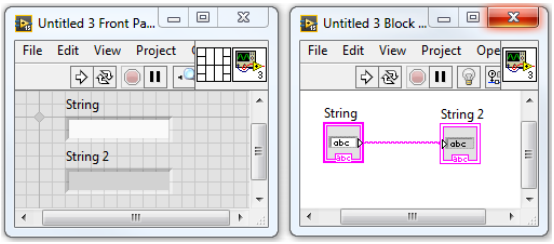

Figure 2.4: String control and indicator example. The connecting wire is not solid as for numeric data, but wavy.

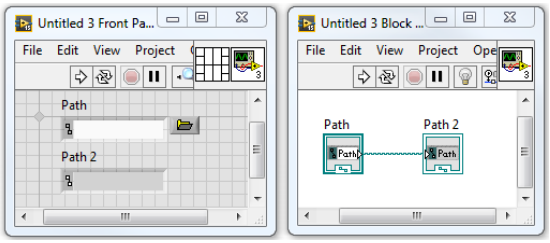

Figure 2.5: Path control and indicator example. The connecting wire is dark green and wavy.

2.2 Arrays

The discussed data types contain only single values. Many processes, such as data acquisition, however, require **array** structures. An array is a group of data elements of the same type. Each element is given an **index number** starting at 0 up to $n-1$ with n being the length of the array. LabVIEW has extensive 1- and multi-dimensional array capabilities.

2.2.1 Creating an array

An array can be created in several ways. If the array is going to be used as a control on the front panel, it can be created as follows: first create the elemental data type. This can be a numeric, string, boolean, etc. Then go to the Array, Matrix & Cluster tab on the controls palette, choose array and drop on the front panel (figure 2.6). You will see that the block diagram representation is black. This means that the array is not yet assigned a data type. Now drag and drop the elemental data type in the array on the front panel. The block diagram representation will get the same color as the element.

The created array structure shows only one element. The index of that element is shown on the left. Changing the index will bring up the accompanying element. Since the first element is grayed out at first, the array has 0 elements. Elements can be added simply by typing in values.

If you want to display more than one element at a time, than hover the mouse over the bottom or the right of the array. A blue expansion dot will appear and the array can be pulled down or to the right to show the 1D data in a column or row respectively. The index number will now indicate the first element that is being shown. Figure 2.7 shows an array of three numeric elements that has been expanded into a column and an array with the same elements that has been expanded into a row. The row array furthermore shows only the elements starting from index 1 (the first element '5' does exist). Note that this expansion only affects the representation on the front panel. It will have no effect on the operations that are performed on the array.

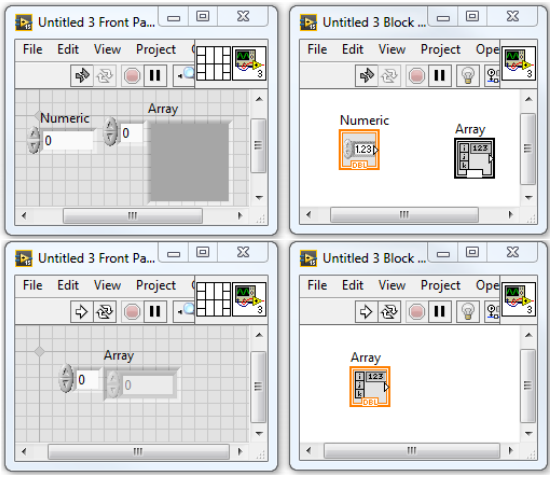

Figure 2.6: Creating an array of double numerics. 1) Create the element (double) and create an empty array. The empty array will have a black color on the block diagram. 2) Drag and drop the numeric in the array on the front panel. The array will now have the same color as the element on the block diagram.

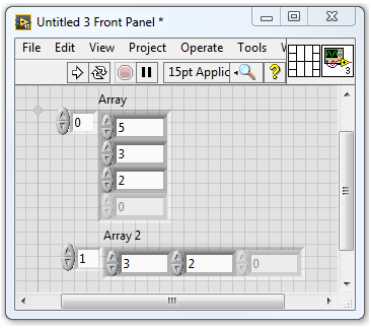

Figure 2.7: A 1D array can be expanded into a column or a row. The index number on the left indicates the first element of the array that is shown. As can be seen the first element of the row array shown is the element with index 1. Grayed out places are non-occupied.

The wires carrying array data are thicker than their elemental equivalents,

so that they can be easily distinguished as can be seen in figure 2.8. An overview of more wire appearances can be found in figure 2.9.

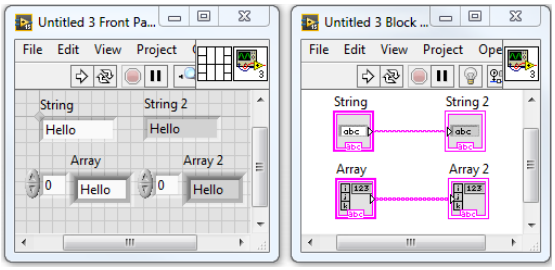

Figure 2.8: Example of the difference in wire thickness for a string element compared to a string array.

Figure 2.9: Overview of common wire types.

Starting from LabVIEW 2015 you can also create an array directly from an elemental data type by right-clicking the element and selecting *Change To Array*.

An array can also be created programmatic on the block diagram by using the function **Initialize Array** that can be found under Programming → Array. This method creates an array of a defined size with elements with the same value (figure 2.10).

Another method for creating an array on the block diagram is to use the function **Build Array**. This function can be expanded so that you are free to add as many elements as you want.

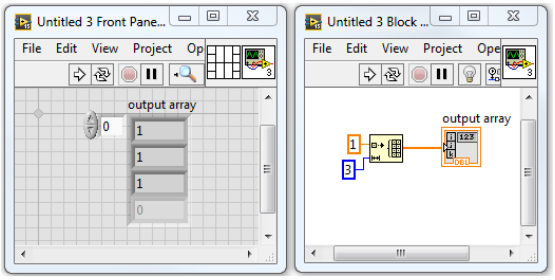

Figure 2.10: Create an array from the block diagram with a specified length and containing elements which all have the same value.

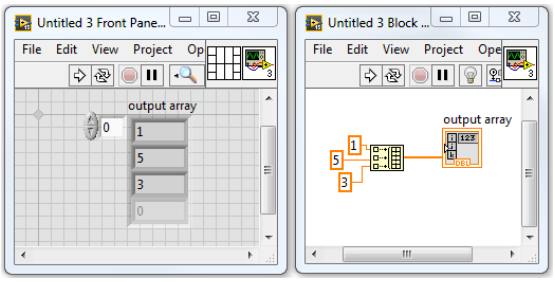

Figure 2.11: Create an array the expandable build array function.

2.2.2 Multidimensional array

A 2D array can be created from a 1D array by right-clicking the index and selecting *Add Dimension*. A new index number will appear (upper: rows, lower: columns) (figure 2.12) and the array can now be expanded in two directions. This 2D array can be turned into a 3D array the same way and so on. There is no immediate limit to the dimension of an array, but in most cases you will only use 1D, 2D and 3D arrays.

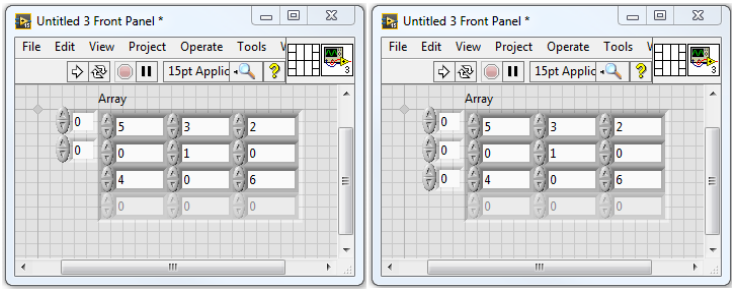

Figure 2.12: 2D and 3D arrays. The 2D array can be represented on the front panel by expanding the array in two directions. The 3D array cannot be completely shown and the indexes need to be used to navigate through the array.

The same way a multidimensional array can be decreased in dimension size by clicking the index and selecting *Remove Dimension*. Take into account that this can lead to the loss of data.

2.2.3 Common Array Operations

In order to use arrays effectively a few important array operations are presented. They can be found on the block diagram under Programming → Array.

The first method is used to determine the size of the array: **Array Size** (figure 2.13).

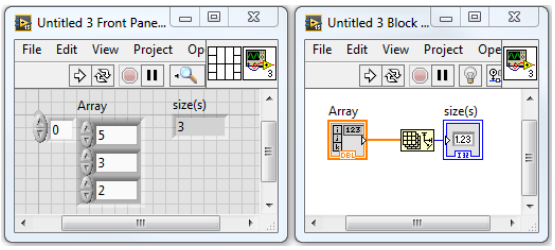

Figure 2.13: Function to get the size of an array. If the array is multidimensional the output will be a 1D array of the sizes of all dimensions.

The second method is to get one element out of an array by its index: **Index Array** (figure 2.14).

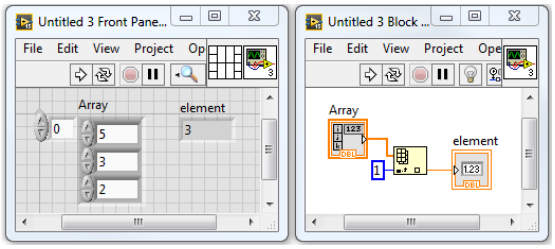

Figure 2.14: Get an element out of an array by giving its index number. In this case the element with index 1 is selected.

The third method replaces the selected indexes with new elements: **Replace Array Subset** (figure 2.15). This method can also be used to replace a sub array with an array. This can be achieved by simply wiring an array to the new element input.

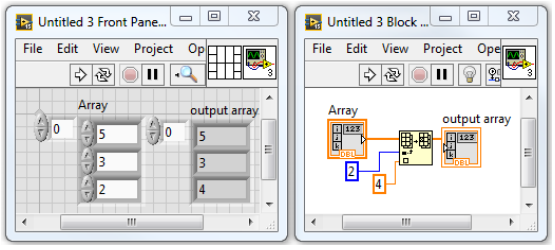

Figure 2.15: Replaces elements in the array at a certain index number with new elements.

The fourth method inserts an element at the specified index number: **Insert**

Into Array (figure 2.16). Again this function can also be used to insert an array as subset.

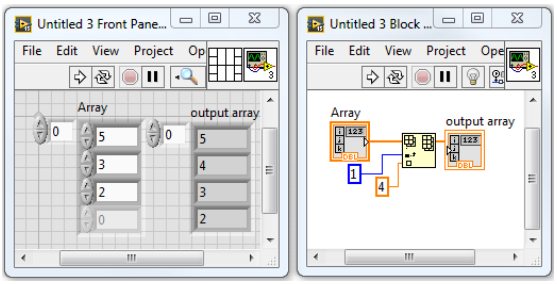

Figure 2.16: Inserts elements in the array at a certain index number. The array will therefore grow in size.

The fifth method deletes a certain length of elements from the array starting at a certain index number: **Delete From Array** (figure 2.17).

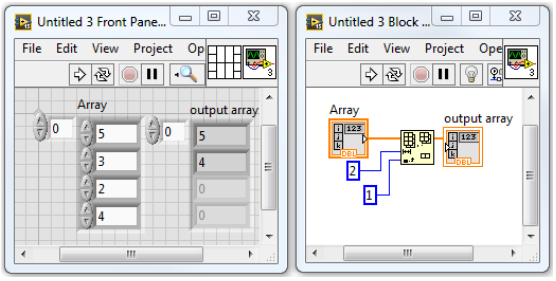

Figure 2.17: Deletes a certain length of elements from the array starting at a certain index number. Therefore the array will decrease in size.

A last interesting operation gets a subset of the original array starting from a certain index number and a defined length: **Array Subset** (figure 2.18).

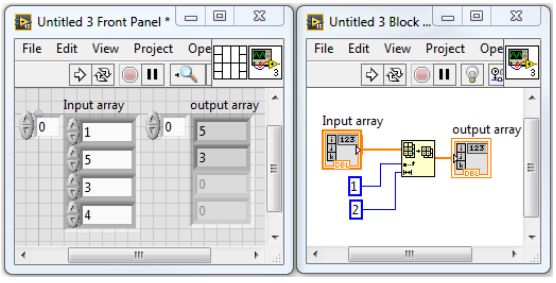

Figure 2.18: Get a subset of the existing array starting from a certain index number and with a certain length.

2.2.4 Numeric operations on arrays

If arrays are composed of numeric elements, several numeric operations can be performed on them as well. If the numeric operation is between an element and an array, the operation will be performed with every element of the array separately (figure 2.19).

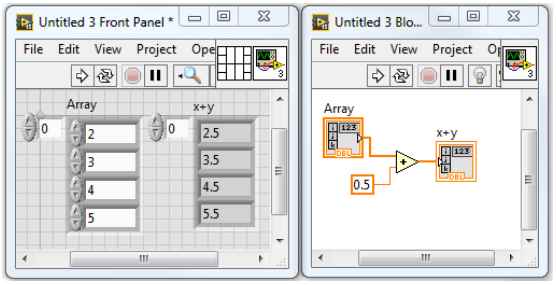

Figure 2.19: Add an element to an array. The result is an array with the operation performed on each element.

If the operation is between two arrays than it will be performed between elements of the same index (figure 2.20). This means that if the arrays have different lengths, the resulting array will have the same length as the smallest array.

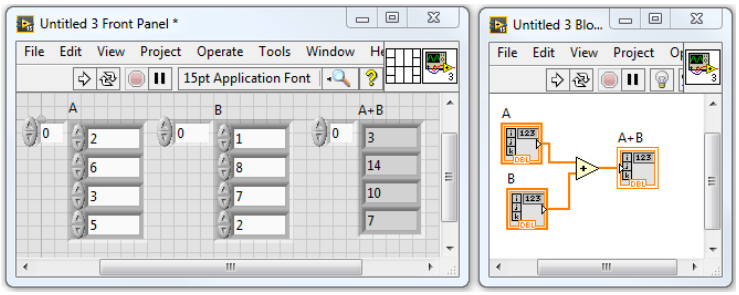

Figure 2.20: Add an array to an array. The result is an array with the operation performed on elementswise.

2.3 Clusters

Clusters like arrays are used to group data. There are however some distinct differences. While an array groups data of the same type, clusters can group different data types, like a numeric together with a string and boolean. Another difference is the fact that there are no operations you can perform on all elements like you would with an array. A cluster is a tool that can help increase the readability of your block diagram by reducing the number of wires running through the block diagram.

2.3.1 Creating a cluster

A cluster can be created much in the same way an array can be created on the front panel (figure 2.21). First create the elemental data types that you want to add to the cluster. Then go to the Array, Matrix & Cluster tab on the controls palette, choose cluster and drop on the front panel. You will see that the block diagram representation is black. This means that the cluster is not yet assigned any data types. Now drag and drop the elemental data types in the cluster on the front panel. If the cluster is purely comprised of numeric data types it will have a brown color on the block diagram. In other cases it will be pink. The cluster can pass all data with one wire (figure 2.22).

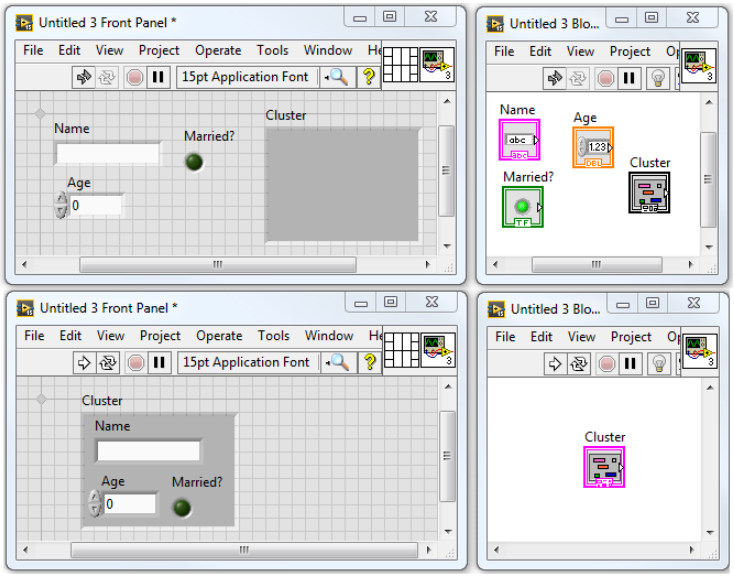

Figure 2.21: Creating a cluster with a string, a numeric and a boolean. 1) First create the different elements and an empty cluster. 2) Then drag and drop the elements into the cluster.

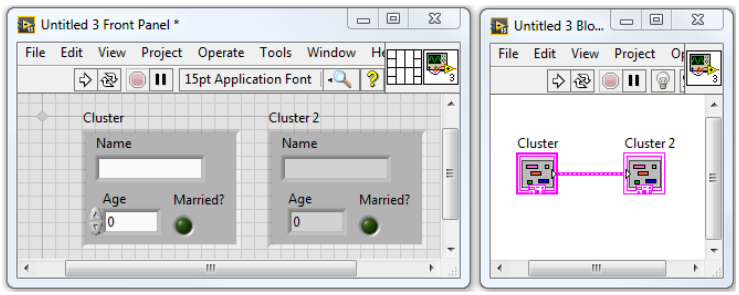

Figure 2.22: A cluster can pass data with one wire.

2.3.2 Using a cluster

The cluster can carry grouped data, but once it comes down to using the elements you need a way to separate the cluster into its components again. This can be done by using the Unbundle by Name and Unbundle functions that can be found under Programming → Cluster, Class & Variant. The Unbundle by Name function is expandable and you can choose the element you want to read by clicking and selecting (figure 2.23).

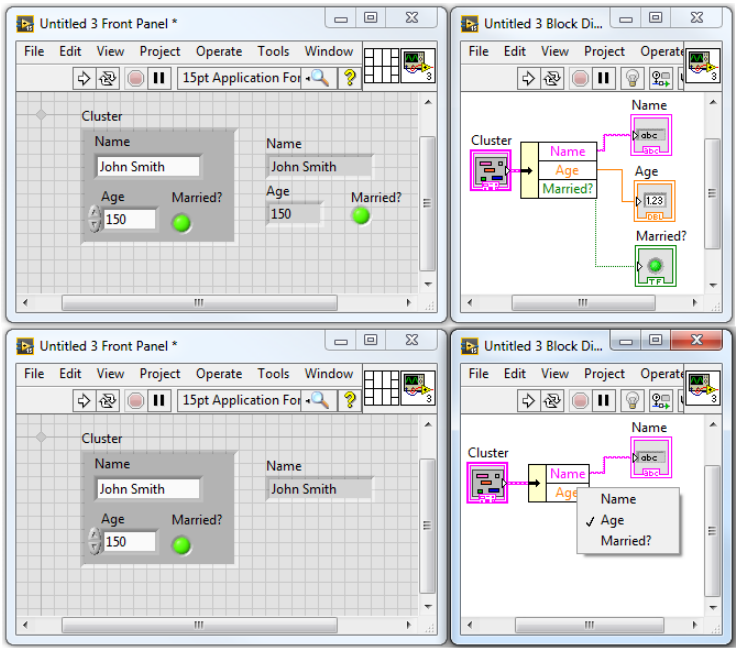

Figure 2.23: The unbundle by name function allows to extract the different elements that make up a cluster.

The Unbundle function extracts all elements from the cluster at once and outputs them in the order they were added to the cluster (figure 2.24). It does not identify the items by name. You can use this method if you want a compact form since Unbundle by Name can take up some space when item names are long.

In order to put information back into the cluster the Bundle By Name (figure 2.25) and Bundle (figure 2.26) need to be used. These function operate in the opposite way compared to their Unbundle equivalents. In order to use these functions an input cluster needs to be wired to the top. The Bundle function is like the unbundle function non expandable and always exposes all elements to the block diagram. When certain values are not wired the values from the input cluster are passed to the the output.

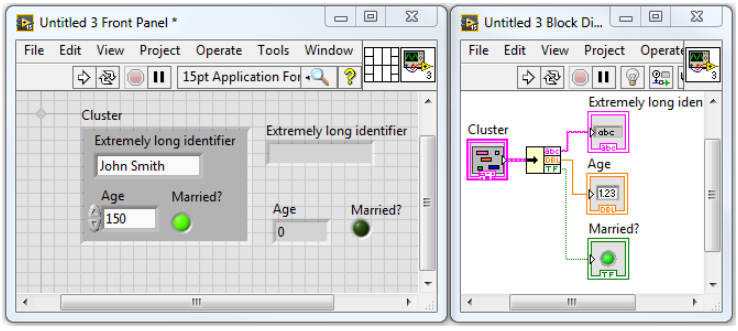

Figure 2.24: The unbundle function allows to extract the different elements that make up a cluster.

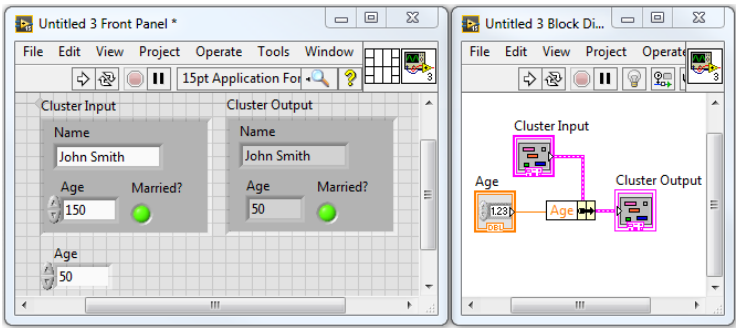

Figure 2.25: The bundle by name function allows to input values for the different elements that make up a cluster.

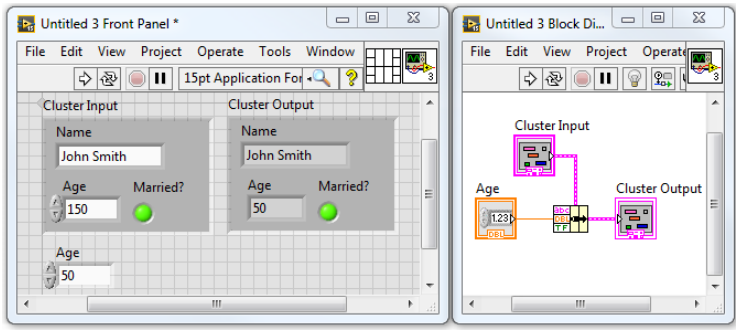

Figure 2.26: The bundle function allows to input values for the different elements that make up a cluster. Input values that remain unwired will take the values from the input cluster.

2.4 Property Node

During runtime properties of controls and indicators can be changed by using property nodes. These properties include color, visibility, format. To use the property right-click the block diagram representation and select Create → Property Node and then select the property you want to get or set. Then drop the

property node on the block diagram. by default the property node will read the state of the property. In order to set the property right-click the node and select Change to Write. Now the property node can be used to set the property.

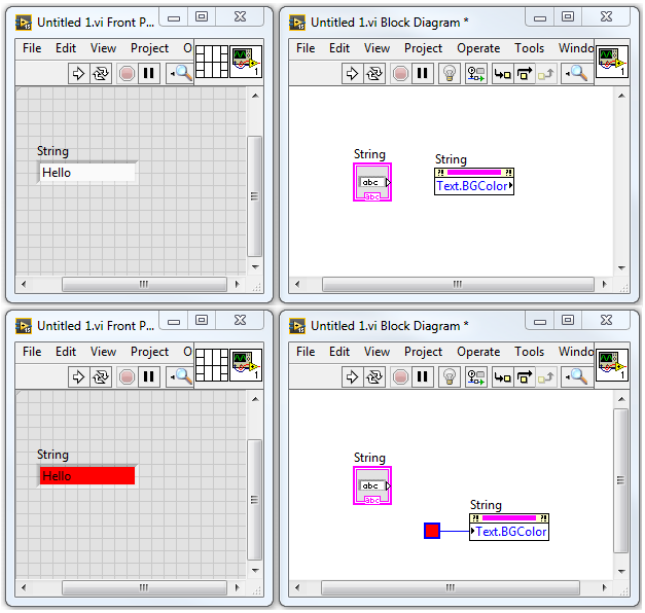

Figure 2.27: The property node allows you to programmaticaly read and write different properties of front panel objects.

2.5 Exercises

1. Create a VI that calculates the dot-product, element by element, of the last three elements of two arrays, one with length 5 and one with length 7 with random numeric values. Write your program so that it still works if the length of the array is changed.

2. Create a subVI that takes a cluster of input settings: A device name, a status, a temperature and a serial number. The VI then needs to check whether the input temperature is within the boundaries of 0°C and 50°C and the serial number is between 1000 and 2000. If this is ok then the status should be true. The VI needs to output the entire cluster.

3. Write a VI that calculates the Root Mean Square (do not use the RMS-function) value of a waveform array. The input is a waveform type (sine, square wave, sawtooth, etc.), the frequency and amplitude. The output is a graph of the function and the RMS-value.

Chapter 3

Structures

This chapter shows the basic programming structures that are encountered in each language: while-loops, for-loops and case-structures. This chapter shows one of the advantages of using graphical programming: whereas there is always a risk of misplacing brackets that confine the loop or case structures in text based languages (figure 3.1), the graphical representation of a structure as a box with the relevant code inside, there is no way you can mistake whether a piece of code is inside or outside the structure.

```
int K = 0;
int Sum = 0;
for (int a = 0; a < 10; a = a + 1){
for (int b = 0; b < 10; b = b + 1){
if (a > 3){
K = a*2+1*3;
}
else{
K = a*3+1*2;}
}
Sum = Sum + K;
cout << "value of a is: " << a << endl;
cout << "value of Sum is: " << Sum << endl;
```

```
int K = 0;
int Sum = 0;
for (int a = 0; a < 10; a = a + 1){
for (int b = 0; b < 10; b = b + 1){
if (a > 3){
K = a*2+1*3;
}
else{
K = a*3+1*2;
}
Sum = Sum + K;
cout << "value of a is: " << a << endl;
cout << "value of Sum is: " << Sum << endl;
```

Figure 3.1: Example of possible bracket misplacement in text-based code. In the left example the K integer is added to Sum inside the outer loop but outside the inner loop. On the right this addition takes place in the inner loop.

3.1 While-loop

The while loop is a structure that executes a part of code as long as a certain condition is not met. This can be any operation that results in a boolean value, for example pressing a stop button, comparing a value to some reference, etc. The default setting is that the loop continues as long as its stop condition is false. Figure 3.2 show the block diagram representation of a while loop. The stop condition is situated on the right (stop sign), the symbol in the left bottom corner is the loop counter. The loop counter starts at zero and goes to $n-1$ with n the number of loop iterations.

A while loop will always run at least once, since the stop condition is only evaluated after the code inside the loop is run.

Figure 3.2: While loop (gray box) with stop condition (red circle) and loop counter (green circle).

3.1.1 Create a while loop

A while loop can be created on the block diagram by selecting Structures → While Loop. When you subsequently click the block diagram the top left corner of the loop is pinned. Moving the mouse cursus now shows a dotted line rectangle that indicates the working envelope of the loop (figure 3.3). Clicking the diagram again will create the loop. You can either draw the loop on an empty space of the block diagram or around already existing code. The code will show a broken run arrow as long as the loop condition is not wired in. Remark that this does not mean that the stop condition will not accept input that can cause the program to get stuck in an infinite loop.

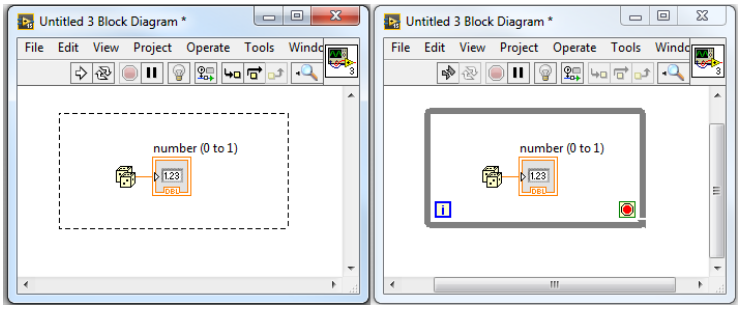

Figure 3.3: Drawing of a while loop (left) and a finished while loop (right). The program will not run as long as the stop condition is not wired.

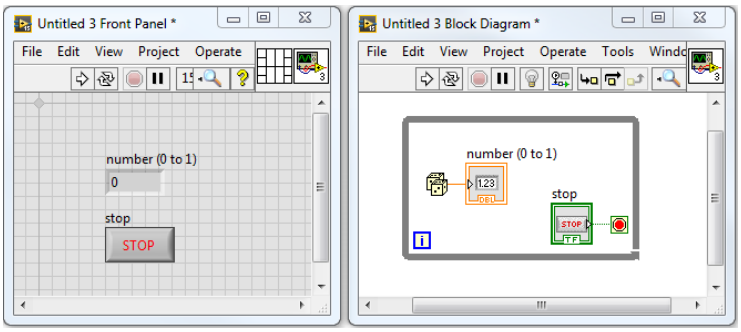

Figure 3.4: Example of a while loop that runs until the stop button is pressed.

Figure 3.4 shows a VI with that generates random numbers in a while loop with a stop condition linked to a boolean button that changes value when pressed.

3.1.2 Inputs and outputs

In order to input values to the loop or output values from the loop a wire can be connected to one of the sides of the loop (figure 3.5). It is best to keep the same convention as for the subVI: inputs go to the left side of the loop, while outputs go to the right side of the loop. Remark that inputs to the loop cannot be changed while the loop is running and outputs cannot be extracted until the loop has finished.

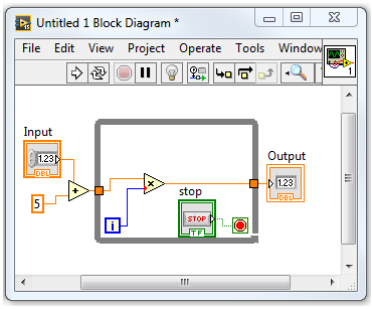

Figure 3.5: While loop with an input and an output. The input value will be equal to the value at the input terminal at the time the loop started (changes will be ignored). The output value will only be available once the loop has finished.

3.1.3 Shift register

In order to pass info from one iteration to the next a shift register can be used. It can be added to the loop by right-clicking the edge of the loop and selecting Add Shift Register. Two black boxes with down and up pointing arrows will appear on the left and right side of the loop (figure 3.6). The left box will input values from the previous loop iteration while the right box will output values to the next loop iteration. For the first iteration the left box will take the value that is wired in from outside the loop (this is called initializing the shift register). When the loop has finished the right box will output a value that can be used outside the loop. The black color means that the data type for the register has not yet been determined. Once data types are being wired to the shift register it will turn into the according color (figure 3.7).

It is important to note that it is bad practice to not initialize a shift register. If no value is wired to the shift register input the loop will take the value from the previous execution and results can differ from what you expect.

Using shift-registers to retrieve data from more than just the last iteration (2nd last, 3rd last, etc.), you can use stacked shift registers. To create a stacked shift register right-click the left terminal (down-pointing arrow) and select Add Element (figure 3.8). An additional terminal will appear below the original

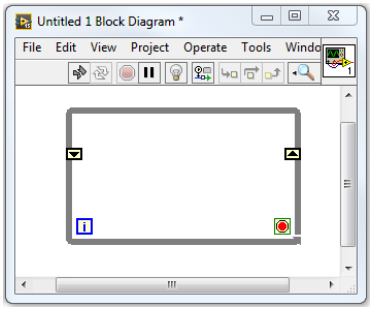

Figure 3.6: Add a shift register to pass data from one loop iteration to the next. The left box with down pointing arrow signals info coming from the previous iteration. The right box with up pointing arrow signals info passed to the next iteration.

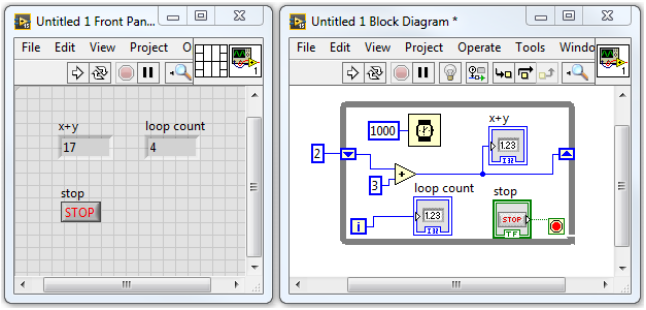

Figure 3.7: Example of the use of a shift register. An integer value of 2 is wired into the loop. This value is added to 3. The resulting value of 5 is then passed to the next iteration. Thus for the second iteration the value of 5 is added to 3 and the result of 8 is passed to the next iteration and so on.

terminal. This terminal will now input the value from the iteration before the previous iteration (figure 3.9).

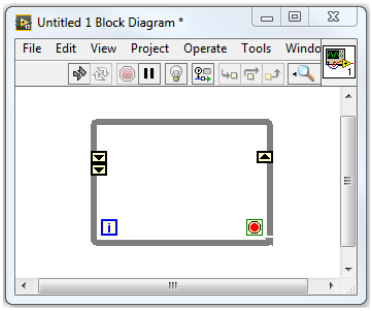

Figure 3.8: Stacked shift registers expose multiple left terminals to the loop so that data from multiple previous iterations can be used.

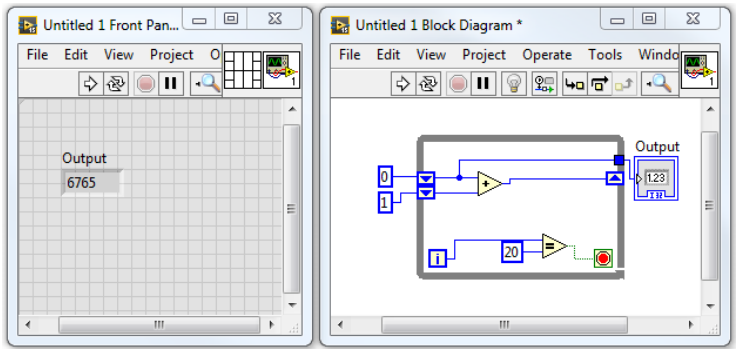

Figure 3.9: Example of a loop calculating the number of the 20th number in a Fibonacci row.

3.1.4 CPU usage

The while loop is an easy way to make a program run until a stop button is pressed. However some care should be taken. LabVIEW will always execute the loop as quickly as possible. This means that the program could start stressing the CPU. In order to avoid this, you can insert timing functions (Programming → Timing). The wait function (figure 3.10) instructs the part of the code it is situated in to wait for the indicated amount of milliseconds before finishing. Placing this function inside the loop will make every iteration last at least the amount of milliseconds specified. When the wait function is inputted with 100 milliseconds the GUI will still look like it is responding instantly. In chapter 5 other tools for creating responsive GUIs will be shown.

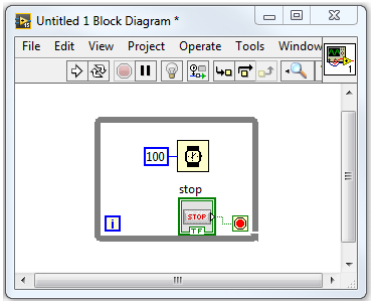

Figure 3.10: While loop with an input and an output. The input value will be equal to the value at the input terminal at the time the loop started (changes will be ignored). The output value will only be available once the loop has finished.

3.2 For-loop

The for loop is a structure that iterates a code segment for a set number of times. Instead of a stop condition that is evaluated every iteration the total iteration count is wired into the loop and cannot be changed during the iteration. Figure

3.11 shows the block diagram representation of a for loop. The total iteration count is situated in the top left corner (N). The for loop has the same iteration counter as the while loop starting at zero and going to $n-1$ with n being the number of loop iterations.

Contrary to the while loop it is possible that a for loop runs zero times. A shift register will in this case pass the input value directly to the output.

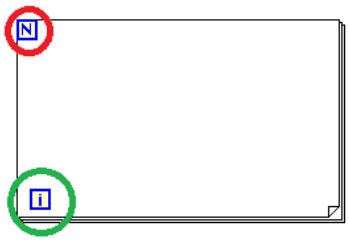

Figure 3.11: For loop (box) with number of total iterations (red circle) and loop counter (green circle).

3.2.1 Create a for loop

The process for creating a for loop is very similar to the creation of while loop. The loop is created on the block diagram by selecting Structures → For Loop. The code will show a broken arrow as long as the total iteration count is not given (unless there is auto-indexing, see next section). Since the total iteration count is of the type Long the for loop cannot be configured to run indefinitely. Again the

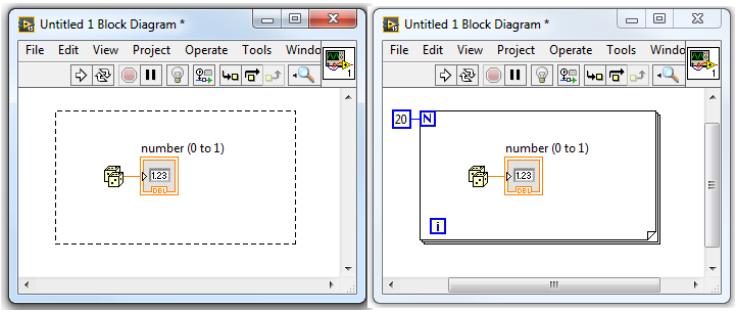

Figure 3.12: Drawing of a for loop (left) and a finished for loop (right). The program will not run as long as the total loop count is not wired.

3.2.2 Auto-indexing

Inputs, outputs and shift registers can be used in exactly the same way as with the while loop. There are however some additional possibilities that exist and are typically used with for loops (note that while loops do have these functionalities as well, but they are less frequently used in the context of while loops) in particular auto-indexing. This means that when an array is wired into a for

loop the tunnel symbol is not filled but has small brackets in it. This indicates that LabVIEW automatically splits the array into its elements and exposes them to the loop one by one with the array index equal to the loop index. In this case the total iteration count does not have to be wired as the loop will run the length of the array (figure 3.13). The auto-index feature is on by default but right-clicking the tunnel will allow you to disable indexing and pass the array into the loop as a whole.

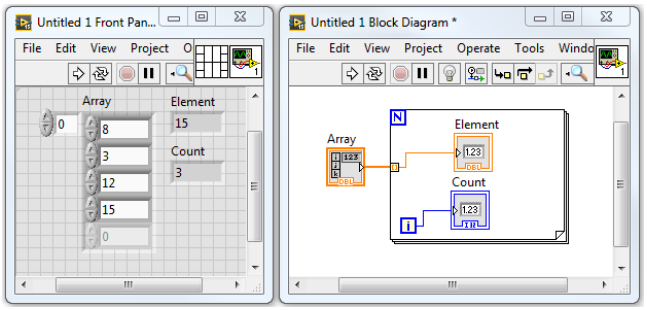

Figure 3.13: Example of the use of the auto-indexing feature. An array wired into a for loop will be split into its elements and presented to the iterations with the array index equal to the loop index.

In the same way elements going out of the for loop can be constructed into an array. For this to happen wire the element to the right side of the loop. The same indexing tunnel symbol will appear and the output is an array. This feature can be switch off by right-clicking the tunnel and selecting Tunnel Mode → Last Value.

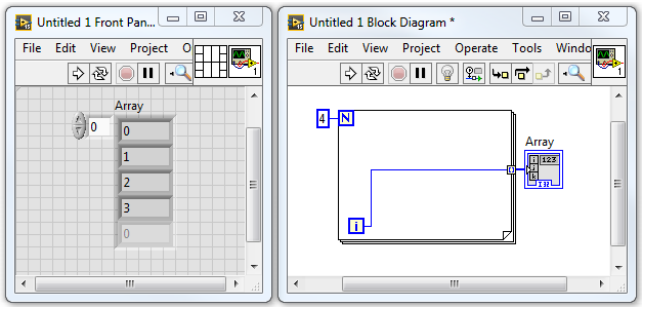

Figure 3.14: Example of the use of the auto-indexing feature for outgoing elements. An element wired out of a for loop will be constructed into an array with the array index equal to the loop index.

3.3 Case-structure

An important part of any program language is the ability to execute specific code blocks only when certain test conditions are met. Most program languages identify this as the if-then, if-then-else and switch-case structures. In LabVIEW

this is known as a case structure. Figure 3.15 shows the block diagram representation of a case structure in LabVIEW. The case structure consists of several substructures. These can be accessed through a menu on the center top of the box. The selector terminal on the center left side of the box selects which case to execute based on the input value.

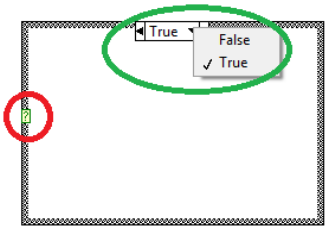

Figure 3.15: Case structure (box) with selector terminal (red circle) and substructure menu (green circle).

3.3.1 Create a case structure

A case structure can be constructed in the same way as a loop. On the block diagram select Structures → Case Structure. The case structure can again be drawn around already existing code. The default configuration of the case structures is a boolean selector and two cases: true and false. This way the structure works the way a If-then-else structure works (figure 3.16).

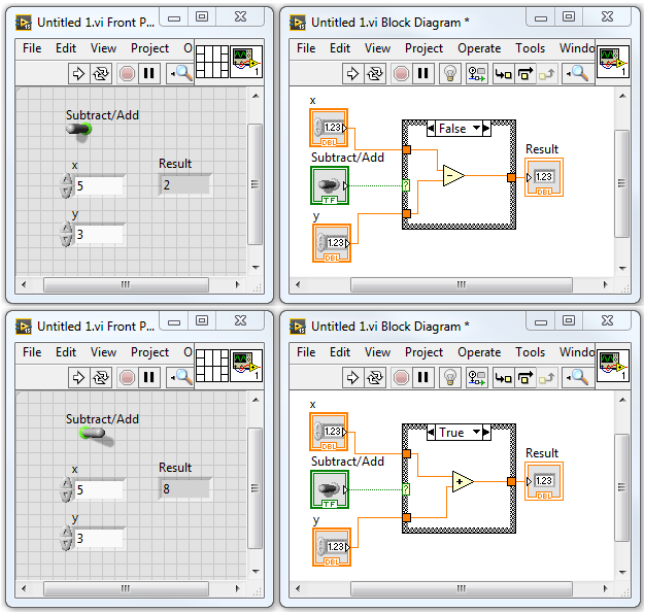

Figure 3.16: Example of a case structure with selector determined by a boolean toggle switch.

Important to notice is that if one case outputs a value, than all other cases must also link a value to that output. Otherwise the tunnel symbol will be open in stead of filled and run arrow will be broken. The same is not true for inputs.

The case structure is however not limited to boolean input. Strings, integers, enumerated types and error clusters can also be used as input. Figure 3.17 shows an example of case structures linked to integer and string controls. Since these types can allow for more than two possibilities a default option has been added to one of the cases. This tells the program to execute that particular substructure when the input is not defined in other cases. The default case can be switched to a different case by right-clicking the structure displaying the preferred case and selecting Make This The Default Case. Left-clicking the text in the selector menu allows you to change the values of a case. Also the number of substructures can be increased. For this right-click the structure and select either Add Case After or Add Case Before (this will determine the order of the added Case in the selector menu).

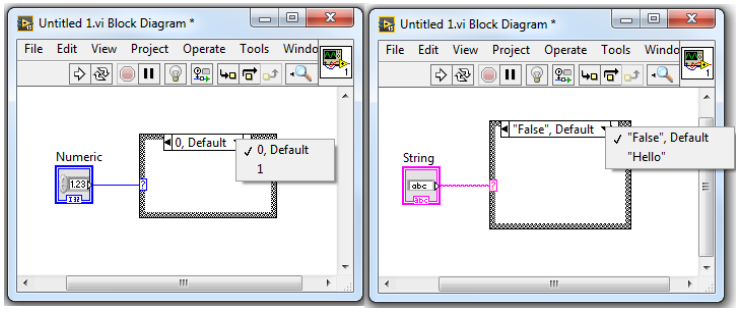

Figure 3.17: Example of a case structure with selector determined by integer and string inputs. A default case has been automatically added to account for inputs not defined in any of the cases.

Cases can be linked to multiple input values (for example one case for 0,1,2 and one for 3,4,5) by using to comma separator. Integer rows can also be implemented: 0..10 equals 0,1,2,3,etc.

3.4 Exercises

1. Use the subVIs you previously created and build a program that allows a user to input a value in degrees Fahrenheit or in degrees Celsius and choose a conversion to degrees Celsius and degrees Fahrenheit respectively. The program must remain responsive for new inputs and may only shut down once the user presses a stop button. Make sure that you program does not overload the CPU.

2. Make a string array. Then make a program that inputs this array and outputs on the one side an array consisting of the 1st, 3rd, 5th, etc. element of the array and on the other side an array that contains the lengths of all the other elements of the original array.

3. Make a control cluster with a numeric, an array and a string. Give these

controls a value. Make a control string that you call Selection. Based on the value of this control show the content of the elements of the cluster:

- Selection = 'Numeric', show the numeric
- Selection = 'Array', show the array
- Selection = 'String', show the string

When the Selection string has an undefined value show the string element of the cluster. Hint: use the property node Visible.

4. Make a succession of tones starting at 150 Hz, with a 0.1 s delay, and 10 Hz step. The program must stop when a maximal frequency of 750 Hz is reached or when a stop button is pressed.

Chapter 4

Error Handling

Error handling is an important part of good programming. It allows for the correct shut down procedures when errors are detected. When working with physical instruments this can be important as these can otherwise get damaged. This chapter will the components of the error cluster, automatic and manual error handling.

4.1 Error cluster

Like other data errors need to be passed along to other subVIs. Therefore there exists a special type of error cluster (figure 4.1). This cluster holds the following information:

- Status: a boolean that is true when an error is detected.
- Error code: the is an integer value that is an identifier of the error.
- Source: string that holds additional information on the exact location of the error.

This cluster can be passed through wires just as other data. Warnings can also

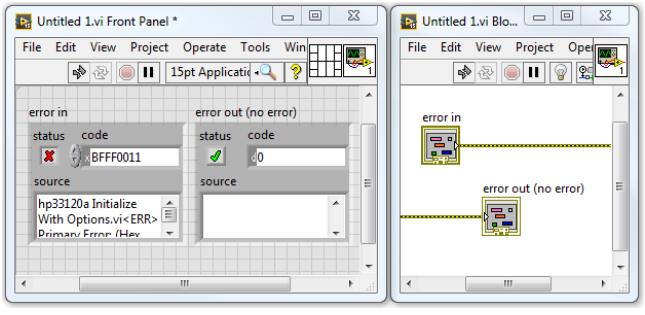

Figure 4.1: Error cluster as a control and as an indicator. Data can be passed through wires. An error can be recognized by the status boolean (true equals error, red 'X'), the accompanying error code and source string.

be passed. These are error clusters with warning code and source string but with a false status.

4.2 Automatic handling

LabVIEW can handle errors automatically. If LabVIEW detects an error it will suspend the execution of the program and display an error dialog box (figure 4.2). The faulty subVI will also be briefly highlighted. The user has the option to continue execution with the error or immediately stop the program.

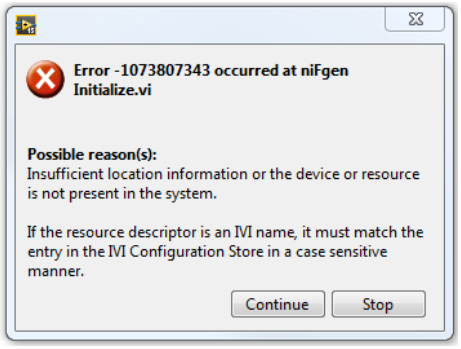

Figure 4.2: Example of an automatically created error message when trying to use initialize hardware that is not present in the system.

In most cases automatic error handling is insufficient. If for example a program controlling an expensive machine that cannot be left turned on for extended periods of time shows an error waiting for a user input while leaving the machine in the on state, damage can occur.

4.3 Manual handling

In order to avoid automatic error handling suspending execution of the entire application, manual error handling should be employed. To do this you can use the error handling VIs together with error clusters. These can be found under Programming → Dialog & User Interface. A lot of predefined LabVIEW functions already feature error inputs and outputs (by convention these are located at the bottom left and right connectors). Figure 4.3 shows an example of a subVI with error inputs and outputs. When these are connected the automatic handling for this subVI will be disabled and errors will propagate through the error wires for further use. When an error enters a subVI it will be passed along. Some functions will cancel execution when an error is wired in. Others will continue to be executed.

4.3.1 Error case

Error lines can be used as selector of a case structure. When an error is wired to the selector the case structure will change appearance. The tabs will automatically change from True and False to No Error and Error with the edge of the

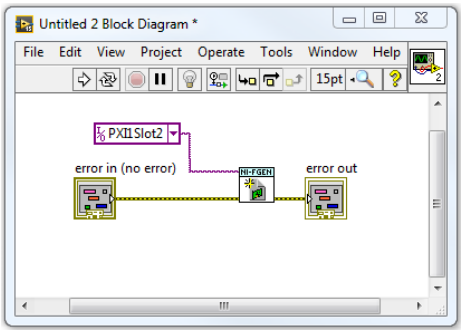

Figure 4.3: Example of a subVI with error inputs and outputs located at the bottom left and right connectors.

structure turning green and red respectively. This case structure is often found in subVIs where you only want to execute the code when no error is coming into the VI and just pass the error along when there is an error on the input.

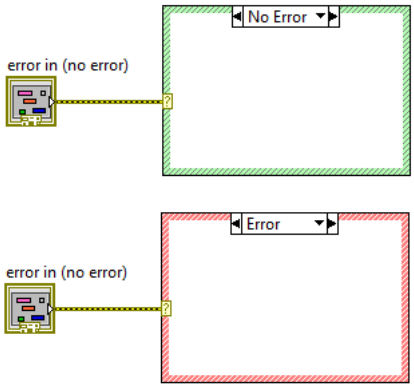

Figure 4.4: Wiring an error cluster to a case structure changes the appearance of the structure to No Error and Error with the edge of the structure turning green and red respectively.

4.3.2 User Defined Error

The user can define errors by filling in the error cluster appropriately. The error codes that the user can freely use are within the range of -8999 to -8000, 5000 to 9999 and 500000 to 599999. The Error Cluster From Error Code VI is a useful tool (figure 4.5) for generating error clusters.

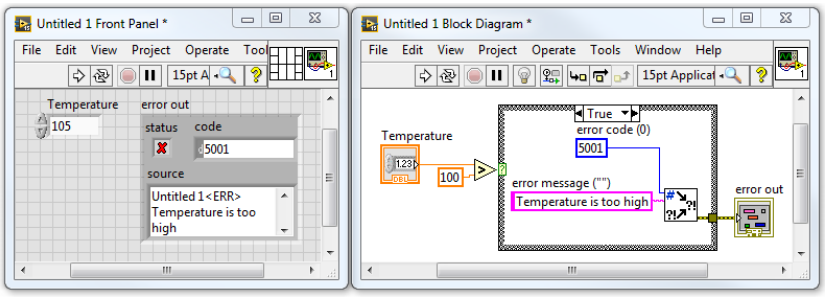

Figure 4.5: The user can define his own errors by using the Error Cluster From Error Code VI. This example generates an error if the temperature control exceeds 100°C.

4.4 Data flow model

Error wires are often used for imposing data flow to different subVIs. As you know by now, LabVIEW executes code based on whether it has received all inputs or not. In some cases however you want a particular part of your code to execute after another part has finished. This can be achieved by packaging the code into subVIs that have error inputs and outputs. Figure 4.6 shows an example of a time delay VI being forced to execute between the execution of two instrument control VIs by connecting the input and output errors because the VI has to obey the data flow paradigm.

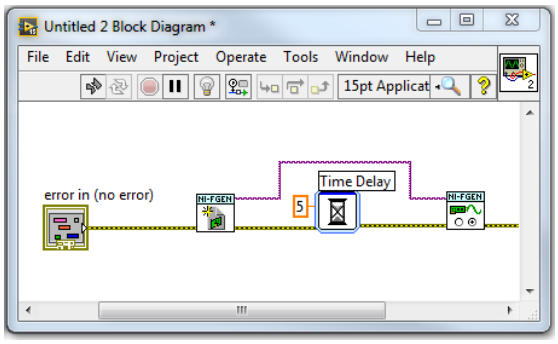

Figure 4.6: Error wires can be used to impose execution order. The time delay VI can only be executed when the initialize VI has executed since the error input from the delay VI comes from the output of the initialize VI. The next instrument VI can only be executed after the time delay VI in the same way.

4.4.1 Merge Errors

In some cases multiple error wires need to be combined. For example different instruments can be initialized in parallel, but a measurement can only be started when all instruments have finished initializing. This can be ensured by combining the error lines of the different initialization processes before starting the measurement. To this end the Merge Errors Function can be used (Pro-

gramming → Dialog and User Interface). This is an expandable function that bundles multiple input error wires into one error output wires (figure 4.7). The function will return the first error found. When no errors are found, the function will search for warnings and return the first warning found.

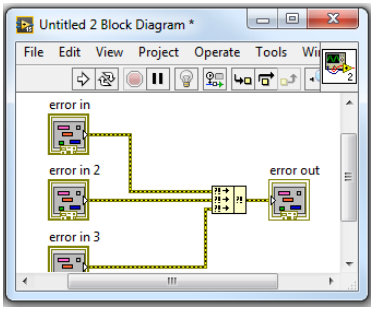

Figure 4.7: Multiple error wires can be merged into a single output error. When more than one error is wired in, the output error will be the first error found. If no errors are found the function will look for warnings.

4.5 Exercises

1. Use the program you previously made for converting values in Fahrenheit to degrees Celsius and add error handling that will generate a user defined error when the temperature value is out of the range $-5°C$ and $60°C$. This error needs to be able to be passed on to the main program that calls the subVI.

Chapter 5

Design Patterns

More complex programs benefit greatly from **Design Patterns**. They are introduced to simplify the development process of these complex programs, because the programmer does not need to reinvent the wheel. Here, three Design Patters, the event handles, the state machine and the Producer/consumer architecture will be discussed.

5.1 Event handler

The **event handler** is used when the program is asking to interact with a user. However, 'asking' an input and receiving nothing, will blow up the CPU. This event structure is already built in such a way that waiting for an event occurs without consuming extra CPU. When an input is registered, it is executed at run time. The event handler is also capable of remembering the order in which multiple inputs happen.

The construction of such an event handler is quite simple and can be made by nest an event structure inside a while loop (figure 5.1). When the operating system broadcasts a system event, such as stroking a key on the keyboard or a mouse click, to the application. This event will be registered and the event structure will execute the appropriate case. When another event occurs during executing a process, the new event will be queued.

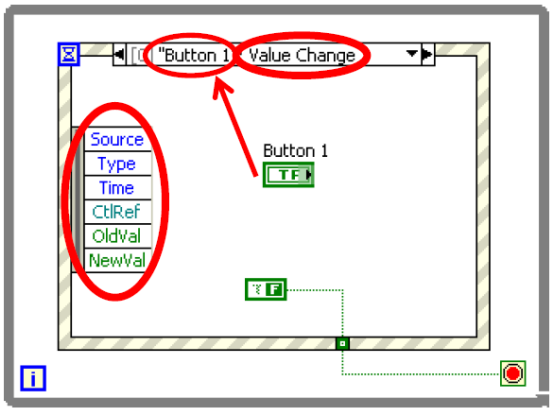

Figure 5.1: The event handler is a design pattern capable to receive multiple commands from the user. These will be processed and the event structure will return the output of the event.

5.2 State Machine

A **state machine** is used when a sequence of events needs to be executed. However, the specific order depends on the input and is determined programmatically. If the sequence were static, the programming was easy and the correct subVIs can just be linked with the next ones. When the program is more complex and depending on the outcome of a subVI, a different subVI can be selected next to execute, even ones that have been activated in the past, the structure is more complex. An example of such a dynamic sequence is found in figure 5.2. A more illustrative example can be found in figure 5.3 in the form of a vending machine.

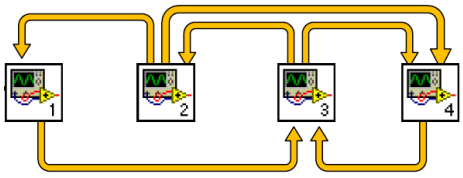

Figure 5.2: Example of a dynamic sequence.

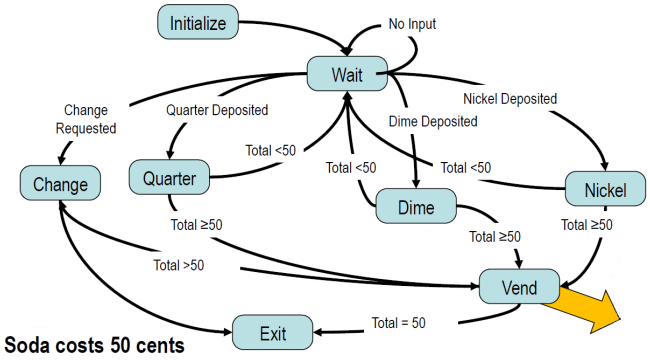

Figure 5.3: A vending machine is a good example of when to use the state machine.

The implementation of the state machine is nothing more that a case structure inside a while loop. Each state has its own case and a shift register carries through the outputted state to the next instance of the while loop. This means that the next routine is determined by the input data. To use the state machine efficient, the use of case structures is beneficial and a table of the possible states can come handy.

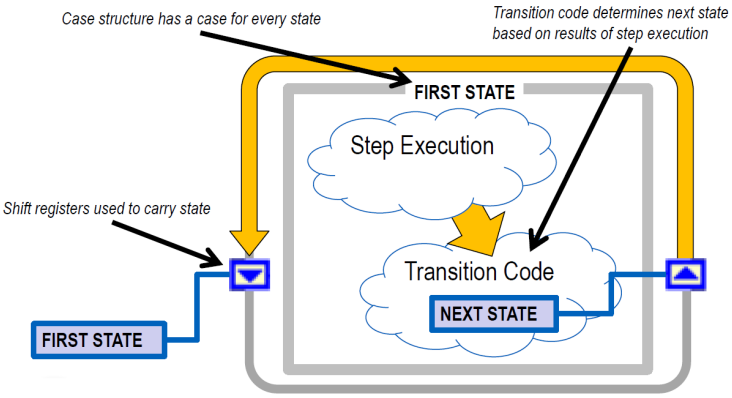

Figure 5.4: Implementation of the state machine.

5.3 Producer/consumer architecture

The last discussed design pattern is the **Producer/consumer architecture**. This architecture is used when two processes need to be executed at the same time, but they may not slow each other down. Most of the time this architecture is used when data is acquired faster than it can be processed. There is one master loop that is always executed, connected to one or more slave loops. They can communicate with one another. The slave loops are only active when data is received from the master loop, or when the queue lists received data. This way the execution of the loops can be asynchronous, because the processes can be

decoupled, from which the user can benefit. Due to this decoupling, each loop will have its own thread.

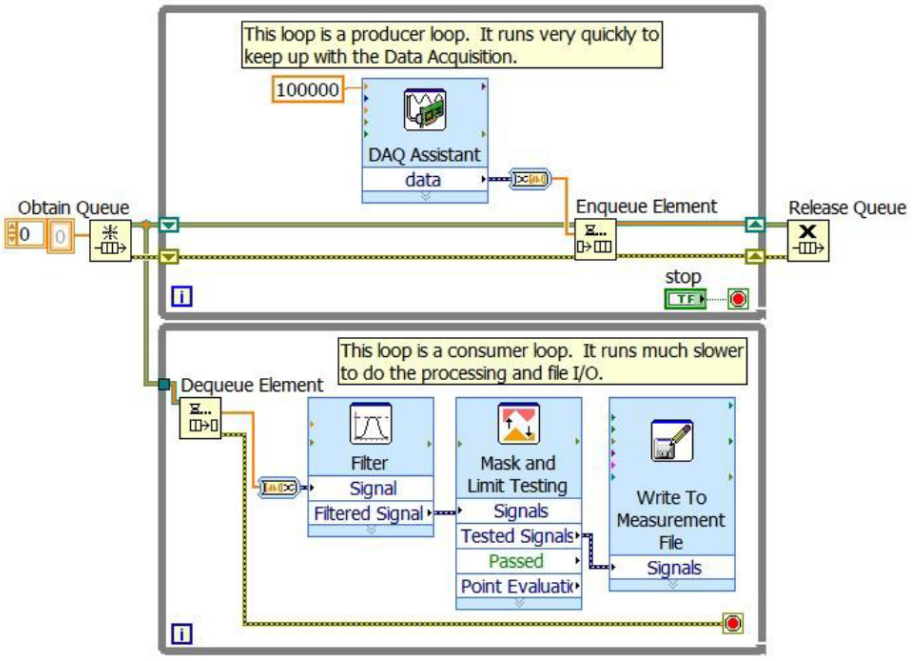

Figure 5.5: Implementation of the Producer/consumer architecture.

5.4 Exercises

1. Build the vending machine with an array of coins as input and the number of sodas and change as output.

Chapter 6

Instrument Control

This chapter discusses the main reason why LabVIEW was brought to life and where it absolutely excels: instrument control. LabVIEW has the capability to connect and interact with a large number of hardware devices, even with a third-party instrument.

6.1 Measurement and Automation Explorer (NI MAX)

Measurement and Automation Explorer (NI MAX) gives you access to all connected instruments and devices. It allows you to investigate the installed software and drivers as well. NI MAX is able to configure the connected devices, save the used configurations, create channels, tasks, interfaces and virtual instruments and update the NI software. You can also fully test every connected instrument using the test panel.

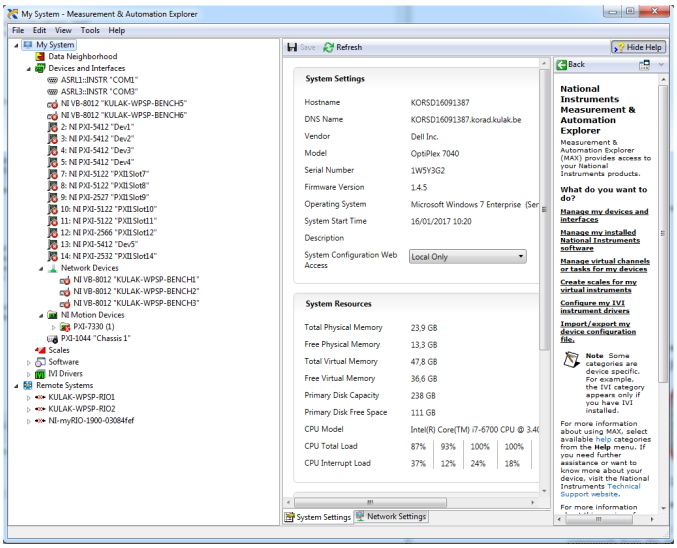

Figure 6.1: Example of a NI MAX screen.

6.2 Instrument drivers

NI MAX is capable of installing the needed drivers itself. Even for most third-party instruments, there exists an instrument driver. These instrument drivers can then be found in the LabVIEW palette 'Instrument I/O'. For example, the instrument driver for the Aligent can be found in figure 6.2.

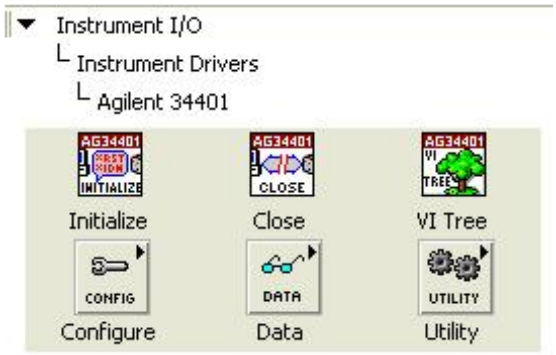

Figure 6.2: Agilent palette

6.3 Error handling

It is important to perform error handling in instrument control applications because there are several potential sources for errors. Error handling with instrument driver VIs is similar to error handling with other I/O VIs in LabVIEW. Each instrument driver VI contains an 'error in' input and an 'error out' output for passing error clusters from one VI to another. The error cluster contains a Boolean flag that indicates if an error occurred, an error code number, and a string that contains the location of the VI where the error occurred.

VISA functions can return errors because VISA or the underlying software or hardware is not properly installed. VISA functions can return errors if the device you are accessing is not responding to the commands you have sent. The instrument could be incorrectly addressed, malfunctioning, or unable to understand the commands that are being sent.

Chapter 7

Project Environment

The **project environment** is a great tool for structuring your LabVIEW applications that contain a large number of subVIs and libraries, that need documentation files, that need to be converted to standalone applications, etc. You can create a LabVIEW project by selecting File → Create Project and then choosing Blank Project. This will open a window as illustrated in figure 7.1.

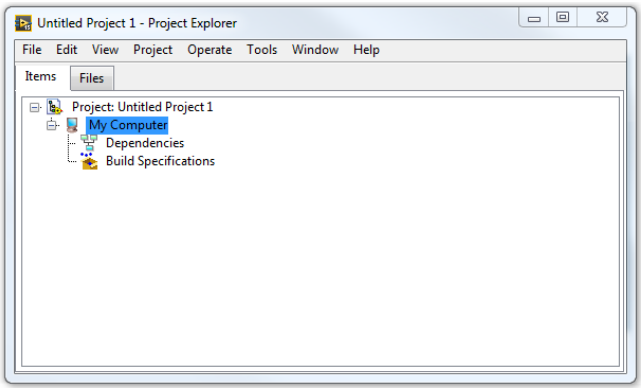

Figure 7.1: A blank project window

7.1 Items and files pages

There are two pages for viewing files in the LabVIEW Project: the Items page and the Files page (figure 7.2).

The default view is the Items page, which provides a tree view of items you have added to the LabVIEW Project and groups them by hardware target (computer, my RIO, etc). You can create folders in this view to either customize how files are organized or to synchronize to a specific location on disk. This is where you likely will spend most of your time. You cannot delete files from disk from the Items page – this is designed to protect users from accidentally deleting code. You can delete items from the Files page only by right-clicking on them and selecting "Delete from Disk."

The Files page shows where items that have a corresponding file on disk are physically located. Use this view to perform the same file operations you would normally perform from within your system file browser, such as moving, copying, or deleting. Because LabVIEW is aware of the changes, it is able to update callers and preserve links when changes are made.

You can customize the hierarchy of files in the LabVIEW Project Explorer without affecting the layout of files on disk; however, it is recommended that the organization in the LabVIEW Project reflect the hierarchy you've set up on disk as closely as possible.

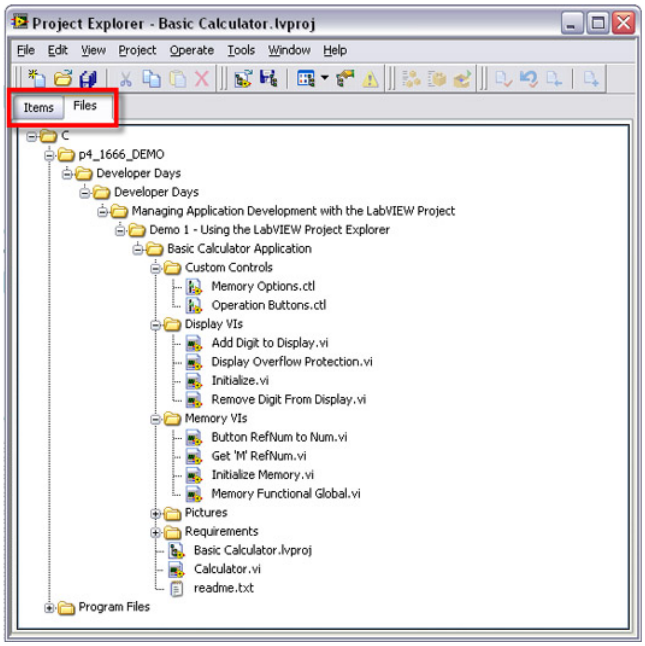

Figure 7.2: A blank project window

7.2 Application builder

LabVIEW is capable of building standalone applications. These make it possible for users to install LabVIEW applications without having a LabVIEW license. The only LabVIEW instance needed is the LabVIEW Run-time Engine, except of course for extra drivers which are used by the application. With simply this installed, the application can run.

Standalone applications are not the only thing the application builder can build. Libraries can also be build from it. These libraries are LabVIEW code that can be called from text-based programming languages. Everything of course needs to be configured correctly, before it can be build. More information can be found on the tutorial website (http://www.ni.com/tutorial/3303/en/) on how to build all applications.

7.3 Organizing items

It is rare for an engineer to work alone when developing a large, complex LabVIEW application. For the vast majority of applications, a team of developers or engineers work together on the same set of code. However, without proper investment in infrastructure and planning, group development can introduce several pitfalls and inefficiencies.

Team-based development often relies on the process of merging a developer's modifications with a master copy of code. Developers commonly develop in branches of the main code base and must merge their branch back into the main code base. Understanding the differences in code can be challenging, especially for very complex code. In addition, merging changes from one code-base to another can be very time-consuming and error-prone.

As an application grows larger, it might become challenging to share and maintain code among teams of developers. Without an in-depth understanding of the complete architecture of an application, developers might not be able to predict the outcome of making a change to a section of code. In addition, applications with monolithic architectures can make it impossible for any of its pieces to be reused in other applications. Finally, even if application components might be reusable, the time necessary to collect and distribute all necessary code might increase development time significantly. To facilitate the reuse and sharing of application components, applications should consist of modular pieces, which developers can run and test independently. The modular pieces should provide a public interface for developers to interface with the component in a defined and controlled manner. In addition, the components should be encapsulated to prevent other developers from working with private instead of public interfaces.

Modularized code can be packaged in multiple ways. In LabVIEW, three of the most common ways of sharing or reusing source code are with project libraries, or .lvlib files, packed project libraries, or .lvlibp and source distributions, or .llb files. Depending on the type of application, each of these libraries has its own advantages over the other and makes the sharing of the source code easier.

Modularized code must expose an interface or API for users to operate on the code's functionality. These APIs expose VIs that can be called by the user, and hide low-level VIs, which perform the expected operations. Developers who wish to give other developers access to the underlying code exposed by the API in addition to the API itself, should use project libraries, or .lvlib files. Project libraries can define a clear public and private interface and provide the library's code to the user. On the other hand, if developers wish to expose an API, but avoid that any user make changes to the underlying code of the library, then they should use packed project libraries, or .lvlibp, which act similarly to DLLs.

Chapter 8

myRIO

The **myRIO** (Figure 8.1) is a portable, reconfigurable, WIFI-enabled device which allows you to investigate on design control, robotics, mechatronics ... from an easy starting point.

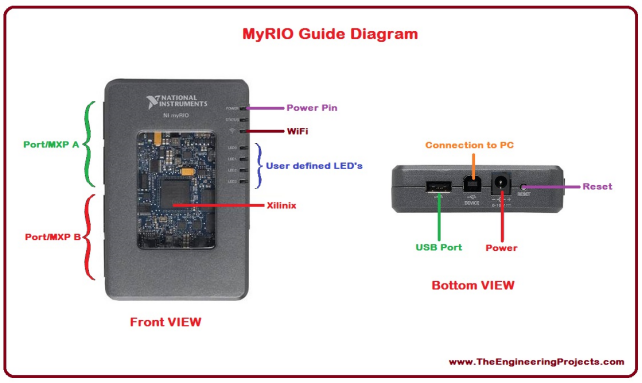

Figure 8.1: myRIO controller

8.1 Installing the software

To program a myRIO, you need to install two extra packages: the Real-Time Module and the myRIO package itself. Log on at ni.com and follow the above links. You need to install the Real-Time module first, then the myRIO module.

8.2 Connector slots

The myRIO providus analog input (AI), analog output (AO), digital in- and output (DIO), audio and power output in a compact embedded device. The myRIO can be connected to a computer over USB and WIFI to upload a program. The connector sides can be seen in Figure 8.2. Connector slotss A and B are identical and are on one side of the device. Connector slot C is on the other side. Each connector in a slot has a predefined function. It is possible to change the connector type.

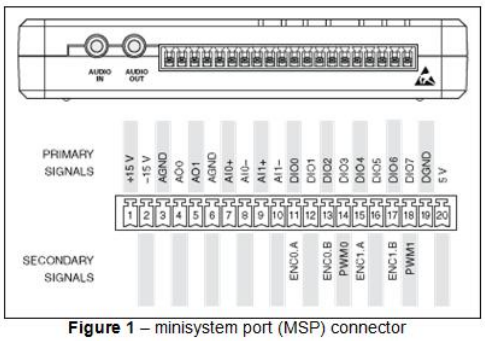

Figure 1 – minisystem port (MSP) connector

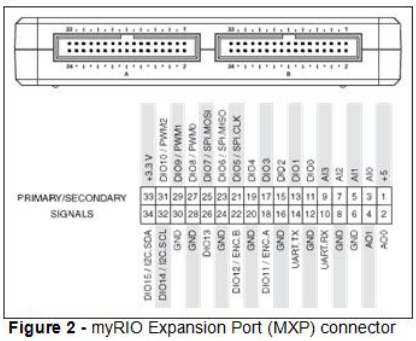

Figure 2 - myRIO Expansion Port (MXP) connector

Figure 8.2: Connector sides of myRIO

- USB: 172.22.11.2
- WIFI: 172.16.0.1 .

By connecting hardware to these connectors, you can read in your sensors or send a signal to your output devices. Before programming, it can be interesting to open the device I/O manager to quickly see how the input looks like or how the output is generated. Once the knowledge is obtained, a myRIO-project in LabVIEW can be started. Just connect your device to a computer and you are ready to go.

8.3 Uploading programs over WIFI

To upload myRIO projects onto the myRIO via WIFI, you need to change the upload-IP of your project to the wireless IP of the myRIO. The standard IPs are: These can be acquired by making connection to the myRIO with a USB-cable,

opening a browser and surfing to 172.22.11.2.

To change the upload-IP, right click on your device, select Properties - General and change to IP Adress to which you want to upload (Figure 8.3). Then change your WIFI-connection to the device's WIFI, with the name 'KULAK-WPSP-RIO#' and password 'KULAK_RIO#' (where # is the number of the myRIO, visible on the labels). You can now upload your programs to the device.

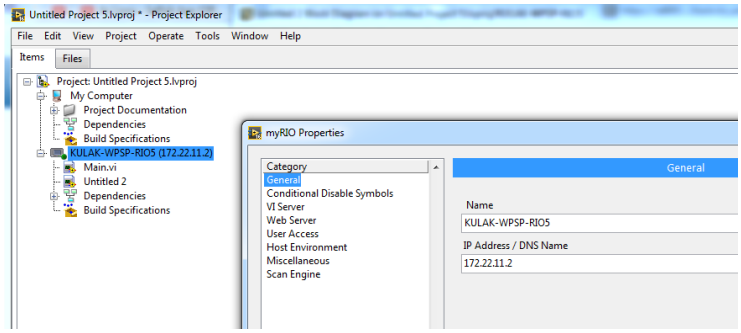

Figure 8.3: Changing the IP to upload your programs to the myRIO

Index

KU Leuven Kulak
Wetenschap & Technologie
Etienne Sabbelaan 53, 8500 Kortrijk
Tel. +32 56 24 60 24
kevin.truyaert@kuleuven.be