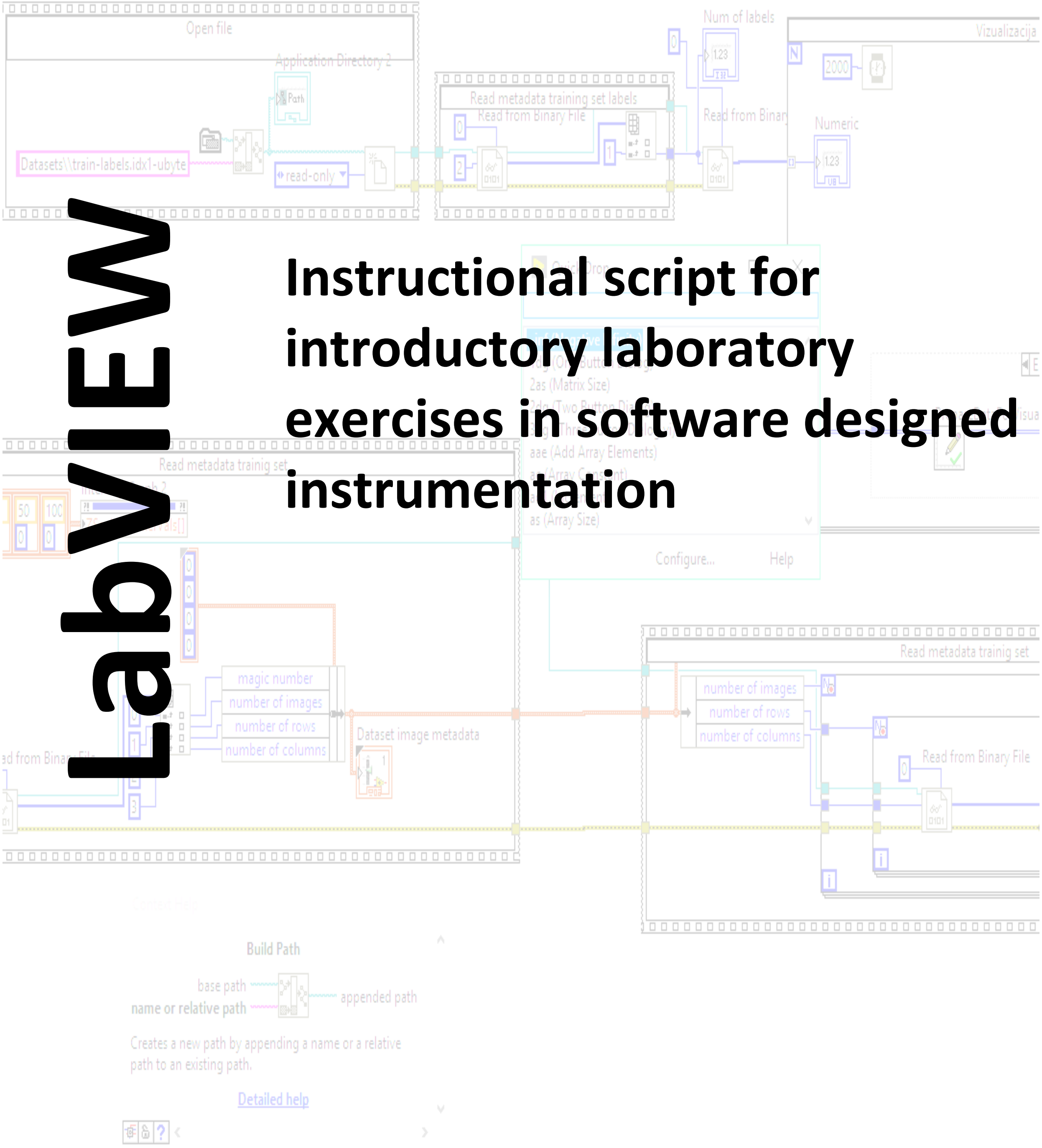
Itasdi

Co-funded by the
Erasmus+ Programme
of the European Union

LabVIEW

Instructional script for introductory laboratory exercises in software designed instrumentation



Creates a new path by appending a name or a relative path to an existing path.

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Table of contents

1. Introduction

LabVIEW is a development environment that enables easier data acquisition and analysis due to its graphical programing capabilities and variety of compatible acquisition hardware.

Some of the key features of the LabVIEW environment are:

- programing of Virtual instruments
- data acquisition using various peripheral devices
- data analysis (filtering, storing, transforming, etc.)
- communication with other systems
- instrument and device control

Programs made in LabVIEW environment are called Virtual Instruments (VI). Their purpose is to simulate a real life measurement device. Virtual Instrument development includes making two main parts:

1. **Front panel** – intended for designing a graphical user interface – part of the system that the users interact with.
2. ***Block diagram*** – adds functionality to the VI. It consists of graphical programing elements like functions, operators, wires, loops, etc.

This document gives an insight into fundamentals of graphical programing and data acquisition using LabVIEW environment. In the second chapter basics of creating and running a VI are explained. Third chapter covers the data types available in LabVIEW. Algorithm flow controls like loops and case structures are explained in the fourth chapter, while the basic data structures, like arrays and clusters are covered in the fifth chapter. Sixth chapter deals with basics of data acquisition. Seventh chapter intends to familiarize the reader to the ways of acquisition hardware simulation. The document is concluded in the final chapter.

2. Creating a new Virtual Instrument

Starting LabVIEW environment a main window appears where the user can choose wheather to create a new project, create a single VI, or open existing project or a VI. Creating a project is useful in cases where there are going to be a lot of dependencies – like multiple dependant VI-s and resurces (for example – files). Wheather the user created a project or not a new VI can be created by clicking in the menu bar: *File->New VI* or using *CTRL + N* keyboard shortcut.

By creating a new VI two windows appear. One window is called the *Front panel* and the other is called *Block diagram.* If user created a blank VI than both of the widows are empty. New elements on both windows are added from the corresponding palette. Front panel elements (buttons, switches, numeric indicators, etc.) are placed from the *Controls palette* – it appears when the front panel is clicked (if not, then it can be opened by clicking in the menu bar View->Controls palette). Block diagram elements are placed from the functions palette.

There is one more palette called *Tools palette* that has all the basic tools available (wire connection tool, operate value, etc.), but, since the tools are by default automatically selected depending on the mouse position, most of the time it can be avoided.

Elements in the palettes are organized in the meaningful groups or sub palettes. For example – if user wants to add a LED indicator to the front panel, it can be found in the Boolean sub palette (*Modern -> Boolean -> Round LED*).

Palettes appearance can be modified by clicking the *Customize* option on the top of the palette and choosing one of the possible visualizations.

User can find the wanted element by navigating through the palette or by clicking the search field on the palette and typing the keyword that corresponds to the wanted element.

By placing an element on the front panel a corresponding block appears on the block diagram. That block is used to retrieve a value from the front panel element or to place a value into the front panel element. If the block diagram retrieves value from the front panel element, then the element is called a Control. If the block diagram places a value in the front panel element then it is called an indicator.

When all the front elements are placed and arranged in the wanted manner the user can switch to adding the functionality to the VI. Functionality is added by connecting blocks, operators, functions, sub-VIs with wires accomplishing adequate data flow.

Example 1

Create a Virtual Instrument with a front panel that consists of two arbitrary switch controls, a LED indicator and button for program termination. The virtual instrument works in such a way that the LED indicator is on (light up) if and only if both switches are on.

Solution

Designing the front panel

Programmer can locate the switch by navigating the palette categories or by using the search option. One of the possibilities is to select the element named Vertical Toggle Switch under the Modern category and the Boolean sub - category. After selection, the element should be placed on the front panel. One can repeat the process for the second switch. After adding the switch, programmer needs to add the LED indicator. It is possible to enter “led” keyword in the search option or to find in in the same category as the switches. Program will also require one button that will serve as a program termination control. The button can be found within the same category of elements. The names of individual elements can be altered by double-clicking on the label above the element. The items should to be positioned so that they are neatly arranged.

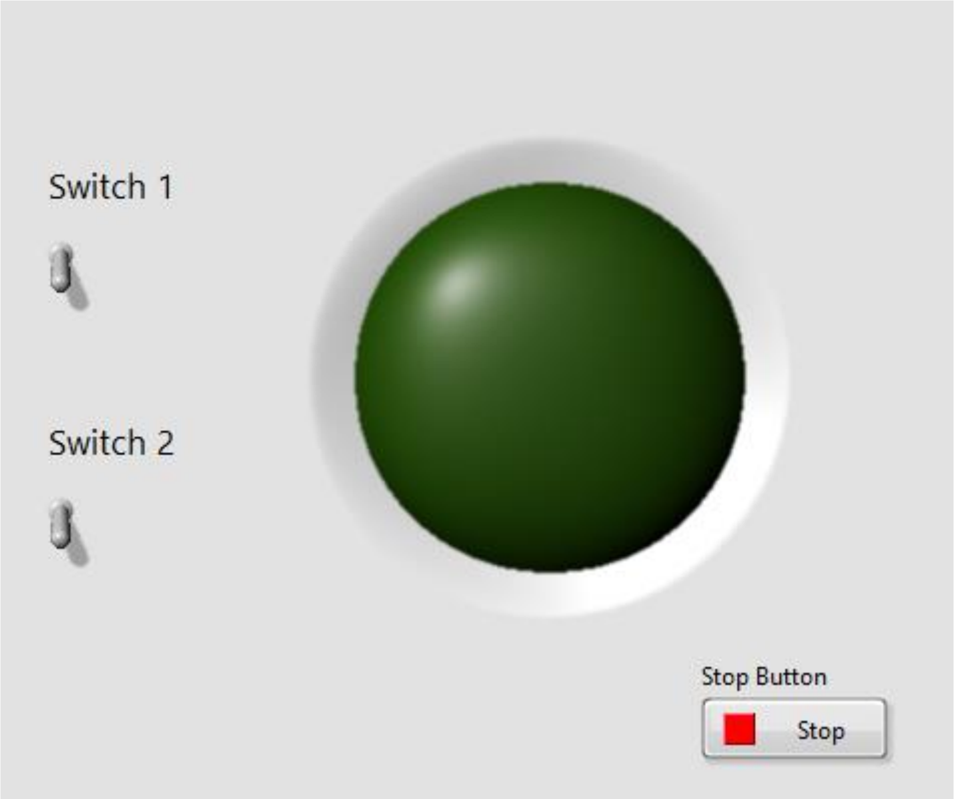


Figure 1 Front panel for the Example 1.

Adding the functionality

By selecting the block diagram window (or pressing the ctrl + E key while the front panel window is active) the functionality implementation of the virtual instrument can start.

Within the block diagram there should be 4 blocks present that correspond to the added front panel elements (two switches, button and LED indicator). When defining the functionality, inputs and outputs of the mentioned elements will be used. In order to implement the desired functionality, it is possible to use a logic operator AND. This element is located within the Programming category -> Boolean -> AND.

The program should be executed until the button on the front panel is pressed. Such functionality will result by adding a loop, for example – a *while* loop. *While loop* can be found within the Programming category -> Structures sub category. After adding the loop structure to the block diagram, it is only necessary to wire the elements in a correct way. It is necessary to ensure that all elements are within the while loop structure.

As mentioned above, the task says the program should end by pressing the button. Such functionality can be achieved by connecting the button output to the input of the *Loop Condition* element that defines when the loop is stopped. The elements inside the loop need to be connected as follows:

● output of the first switch to one input of the AND operator.

● Output the second switch to the other input of the AND operator.

● output of the AND operator to the LED indicator input.

The block diagram should look like the one presented by the Figure 2.

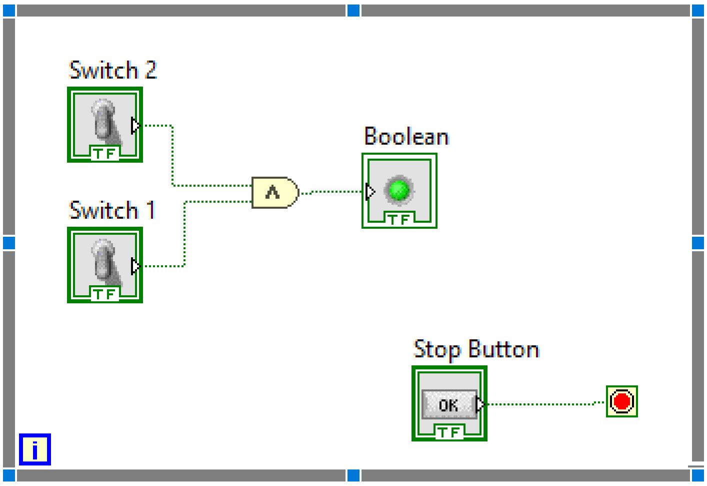


Figure 2 Block diagram of the Example 1.-

Running the program

Execution of the VI is controlled by the Execution buttons shown in the Figure 3 that are placed on, both, the front panel and block diagram. Starting of the VI is done by pressing the Run button located on the toolbar. Stopping the program can be performed by pressing the *Abort Execution* button or when all the operations on the block diagram are done. Pressing the *Run* button program executes once. If the user is required to run the program continuously until the *Abort Execution* key is pressed one can use the *Run Continuously* button. *Run Continuously* works as if the entire block diagram was placed inside the infinite loop. Loops will be explained in the later chapter.

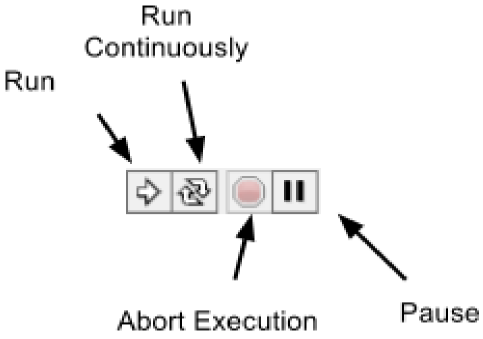


Figure 3 Execution controls

Context help

Context help helps the programmer to work with the LabVIEW environment. It can be turned on by clicking on the icon at the top right corner of the application window.

When the Context help option is turned on, it is enough to move the mouse over the icon or a block and the help is automatically displayed within a special window. This window contains the basic functionality explanation of the element over which the mouse pointer is set. However, for more detailed explanation, programmer can click the link inside that window called *Detailed help.*

Useful keyboard shortcuts in the LabVIEW environment

The Table 1 shown below shows a preview of some of the keyboard shortcuts that are commonly used in LabVIEW environment.

Table 1 Useful keyboard shortcuts

KEYBOARD SHORTCUT	FUNCTION
CTRL + R	Launch VI.
CTRL + .	Stop the VI.

CTRL + B	Erases all unconnected wires.
CTRL + Z	Undo.
CTRL +SHIFT + Z	Redo.
CTRL + S	Saving VI.
CTRL + SHIFT + S	Saving all open VIs.
CTRL + Q	Exit LabVIEW Tool.
CTRL + E	Change focus between the block diagram and the front panel.
CTRL + T	Set the front panel window and block diagram window next to each other.
SHIFT + CLICK	Select multiple objects on the block diagram or the front panel.
CTRL + HOLD LEFT MOUSE CLICK AND MOVE	Creating a blank space on the block diagram or on the front panel.
CTRL + I	Changing VI (icon) properties

Assignment 1

Create a virtual instrument using 5 switches that represent binary digits of an integer number. The virtual instrument should turn on the LED indicator only if the decimal value of the binary number represented by the switches is 21.

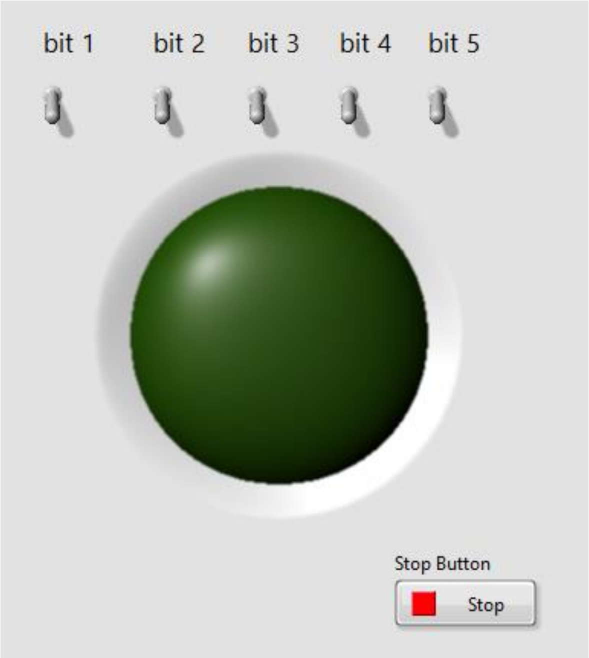


Figure 4 Possible front panel for the Assignment 1 VI.

3. Controls and indicators

Controls allow data entry in the LabVIEW program (Virtual Instrument - VI), while the indicators allow data to be displayed on the front panel of the VI. Each indicator can be changed by right-clicking on the element and selecting Change to control and vice versa.

Data types

LabVIEW supports multiple data types:

- Numerical data types - integeres, floating point, complex, etc.
- text (strings)
- boolean - logical
- arbitrary – enums, clusters, variant

Wires and elements intended for the string data type are colored in pink, those intended for integers are blue, those intended for floating point numbers are orange , for boolean values in green and for example those intended for values representing time are brown.

Numerical data types

Within the LabVIEW tool, numeric values can be stored or displayed as floating-point numbers, fixed-point numbers, integers, unsigned integers and complex.

Each type of data can save a different type of numeric values and different ranges of these values. The value range most often depends on the number of bits that this data type uses and the way the values is converted to the binary stream. This greatly contributes to the efficient utilization of memory that is available to the program. Within the Table 2 shown below, there are types of data with the corresponding block appearance, number of bits, minimum and maximum values that this type of data can store.

Table 2 LabVIEW basic data types

Terminal	Data type	Number of bits	Minimum value	Maximum value
SGL	Single precision	32	Minimum positive: 1.40e–45 Minimum negative: –1.40e–45	Maximum positive: 3.40e+38 Maximum negative: –3.40e+38
DBL	Double precision	64	Minimum positive: 4.94e–324 Minimum negative: –4.94e–324	Maximum positive: 1.79e+308 Maximum negative: –1.79e+308
EXT	Extended precision	128	Minimum positive: 6.48e–4966 Minimum negative: –6.48e–4966	Maximum positive: 1.19e+4932 Maximum negative: –1.19e+4932
CSG	Complex number – single precision	64	Same as single precision for imaginary and real part.	Same as single precision for imaginary and real part.
CDB	Complex number – double precision	128	Same as double precision for imaginary and real part.	Same as double precision for imaginary and real part.
CXT	Complex number – extended precision	256	Same as extended precision for imaginary and real part.	Same as extended precision for imaginary and real part.
FXP	Fixed – point	64	ovisi o korisniku	
I8	Byte signed integer	8	–128	127
I16	Signed integer - riječ	16	–32,768	32,767
I32	Long signed integer	32	–2,147,483,648	2,147,483,647
I64	Quad signed integer	64	–1e19	1e19
U8	Byte unsigned integer	8	0	255
U16	Word unsigned integer	16	0	65,535
U32	Long unsigned integer	32	0	4,294,967,295

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

U64	Quad unsigned integer	64	0	2e19
⌛	128-bit time	128	Minimalno vrijeme: 01/01/1600 00:00:00	UTC maksimalno vrijeme: 01/01/3001 00:00:00 UTC

After adding an numeric indicator or a numeric control (eg.: Controls -> Modern -> Numeric -> Numeric Indicator), the data type that will be stored / displayed by this block can be changed by right-clicking on the icon and selecting the *Representation* option.

Things like the range of numbers that are allowed to be entered in the numeric control can be set by right-clicking on the control and selecting the Properties option. Within this option, programmer can, also: adjust the appearance of the element, minimum and maximum allowed value for controls, display format for indicators, add a documentation information for the element, etc. Setting a minimum or maximum possible input value on a control can prevent a user from entering a higher or lower value than that allowed by the program – which means that programmer does not have to implement the validation functionality.

Casting of data types

Sometimes you need to convert some data type to another data type. For conversion operations from one data type to another LabVIEW has embedded conversion blocks or the conversion is done automatically. Conversion blocks from one type of data to the other are located within the Functions palette: Programming -> Numeric -> Conversion. Numeric data types can be automatically converted from one to another if output terminal of one type is connected to input terminal of another data type.

String Data Type

A String consist of a stream of characters (often ASCII, or Unicode). String controls and indicators are used to enter, or display, textual data. The indicators and controls used with the string data type are contained within the control palette under the category Modern (Silver or Classic) -> String & Path. String operations and conversion from numeric types to strings can be done using blocks within the Functions palette under the Programming -> Strings group.

Boolean data type

Boolean data type can only take on two possible values true or false. In LabVIEW any control or indicator that can be in two possible states is considered of boolean data type, like various switches and buttons. Boolean indicators and controls are located in the Controls palette within the Modern (Classic or Silver) -> Boolean group.

.

4. Operators

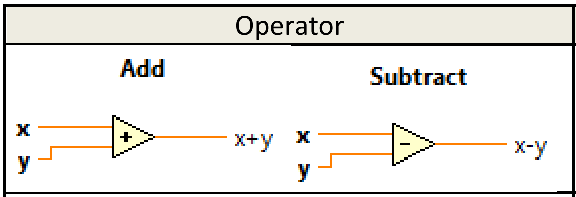
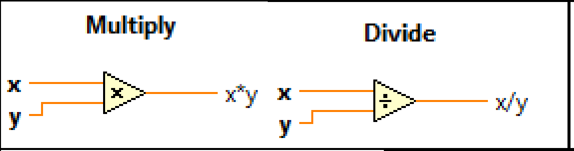
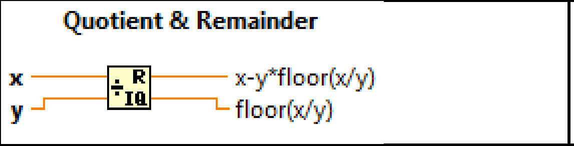
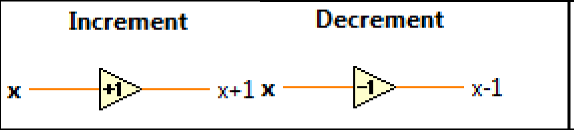
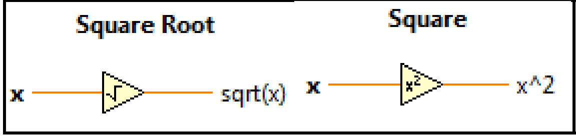
Operators are used to carry out various arithmetic and logic operations on data. Within this document, operators are divided into three basic groups inspired by the groups within the LabVIEW tool itself.

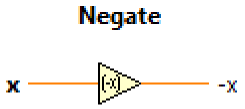
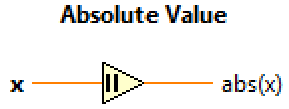
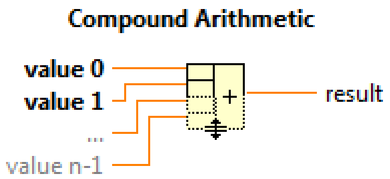
Arithmetic operators

Arithmetic operators serve to perform a series of arithmetic operations over numerical values. Since they are working on numeric values, they are located within the Numeric section under the Programming group as part of the Functions palette.

The basic arithmetic operators and their functionality are given in the Table 3 shown below.

Table 3 Arithmetic operators and their description

Operator	Functionality
	adding / subtracting of input
 Multiply x y × x*y Divide x y ÷ x/y	Multiplication / division of Input Values.
 Quotient & Remainder x y R IQ x-y*floor(x/y) floor(x/y)	Calculating the integer remainder of division, and the result of an integer division.
	Increase value for 1 and decrease value for 1.
 Square Root x √ sqrt(x) Square x x² x^2	Root / square of the input value.

	Changing the sign of the input value.
	Calculation of the absolute value of the input value.
 Compound Arithmetic value 0 value 1 ... value n-1 + result	Performs arbitrary arithmetic operation over multiple input parameters. It is possible to perform the following operations: addition, multiplication, logical "AND" and "OR" and “XOR". Selecting the operation is done by right-clicking on the block and selecting *Change mode*.

Example 1.

Create a virtual instrument that takes on two input parameters a and b - that the user enters. VI should calculate the solution for the following equation::

$$y = 2a^2 + \frac{b}{2} - 6$$

Solution

Designing the front panel

On the front panel programmer needs to add two numeric controls that will allow the user to input parameters *a* and *b*. Location of elements: Controls palette -> Silver -> Numeric -> Numeric Control. Also, a numeric indicator (Silver -> Numeric -> Numeric Indicator) is required to display the equation solution.

Possible Front Panel Layout is shown in the Figure :

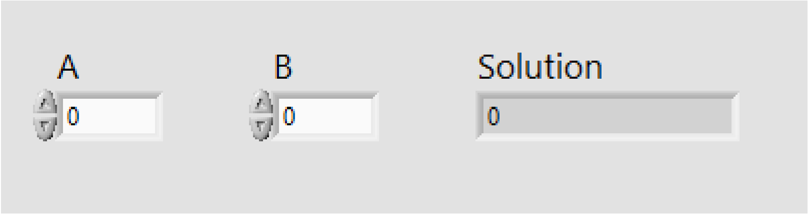


Figure 5 Front panel layout for the Example 1

Adding the functionality

After adding the elements of the front panel, the corresponding terminals appear on the block diagram. Subsequently, adequate functionality needs to be implemented.

In order to achieve the functionality, set by the equation, the input parameter "a" must be squared and multiplied by 2. The easiest way to add the constant 2 is by right-clicking on the input terminal of the multiplication block and by selecting the *Create constant* option. The value of the input parameter "b" is to be divided by 2 and the solution added to the summing operator with the previous result. The value obtained must be reduced by 6 and connected to the numeric indicator so that the solution can be displayed on the front panel.

Block diagram of the Example 1 is shown in the Figure 6.

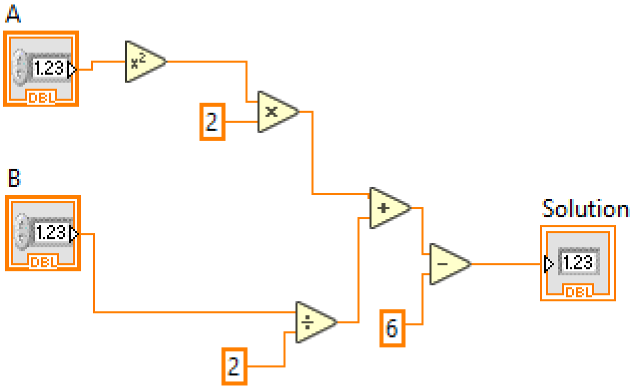


Figure 6 Block diagram for the Example 1

Logical operators

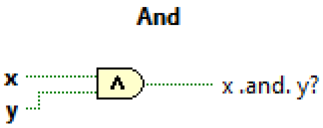
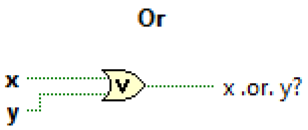
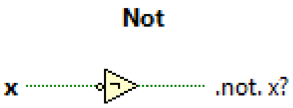
Logical operators perform operations over logical values. There are only two logical values:

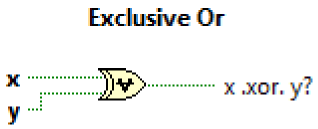
- true (true, YES, 1)
- false (false, NO, 0)

Logical operators, depending on the logical values at the input terminals, produce logical values on the output.

The basic logical operators are listed in the Table 4 shown below and can be found on the Functions palettes: Programming -> Boolean.

Table 4 Logical operators and their description

<table>
<tr><th>Operator</th><th>Functionality</th></tr>
<tr><td>
</td><td>Otuput is true only if both inputs are true.
<table>
<tr><th>x</th><th>y</th><th>output</th></tr>
<tr><td>F</td><td>F</td><td>F</td></tr>
<tr><td>F</td><td>T</td><td>F</td></tr>
<tr><td>T</td><td>F</td><td>F</td></tr>
<tr><td>T</td><td>T</td><td>T</td></tr>
</table></td></tr>
<tr><td>
</td><td>Output is true if at least one input is true.
<table>
<tr><th>x</th><th>y</th><th>output</th></tr>
<tr><td>F</td><td>F</td><td>F</td></tr>
<tr><td>F</td><td>T</td><td>T</td></tr>
<tr><td>T</td><td>F</td><td>T</td></tr>
<tr><td>T</td><td>T</td><td>T</td></tr>
</table></td></tr>
<tr><td>
</td><td>Negates the input.
<table>
<tr><th>x</th><th>output</th></tr>
</table></td></tr>
</table>

<table>
<tr><td></td><td><table><tr><td>F</td><td>T</td></tr><tr><td>T</td><td>F</td></tr></table></td></tr>
<tr><td>
</td><td>True is on the output if the same logical values are on the input.<table><tr><th>x</th><th>y</th><th>output</th></tr><tr><td>F</td><td>F</td><td>F</td></tr><tr><td>F</td><td>T</td><td>T</td></tr><tr><td>T</td><td>F</td><td>T</td></tr><tr><td>T</td><td>T</td><td>F</td></tr></table></td></tr>
</table>

Example 2

Implement a virtual instrument that turns on the LED indicator on the front panel based on the state of 4 switches also on the front panel. The LED indicator light up only if the following switches are ON:

- 1, 3 and 4
- or
- 1, 2, and 3.

Solution

Designing the front panel

Possible front panel layout is shown in the Figure 7. Most of the elements can be found in the Modern -> Boolean sub palette.

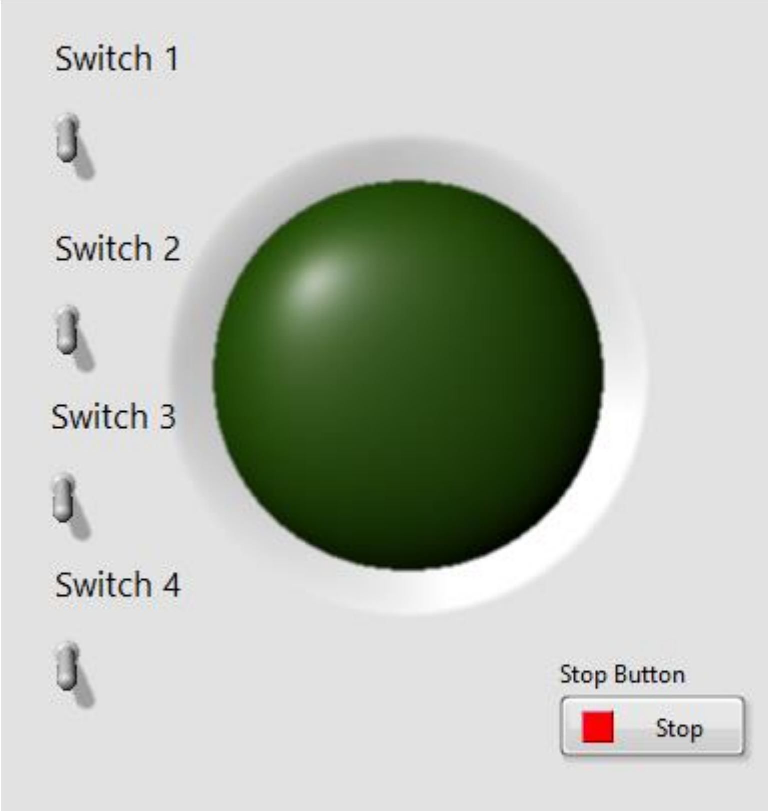


Figure 7 Possible front panel for the Example 2

Adding the functionality

The solution shown in the Figure 8 is using a block called "Compound arithmetic" that has the option to perform logical "I", "OR", "XOR" over multiple inputs.

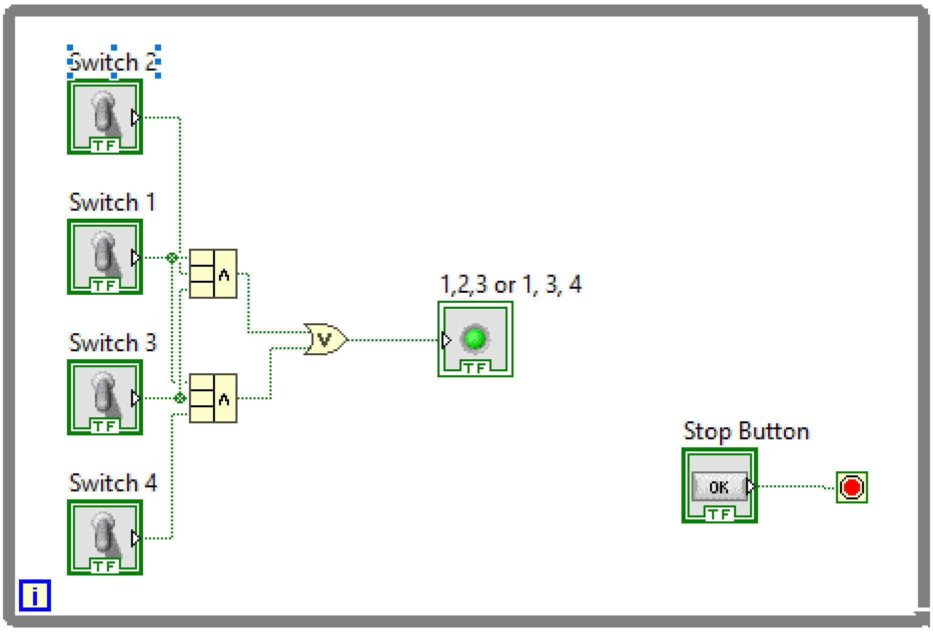
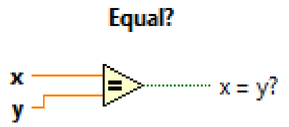
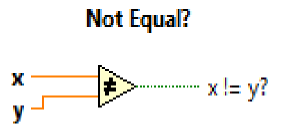


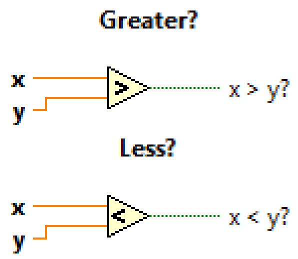
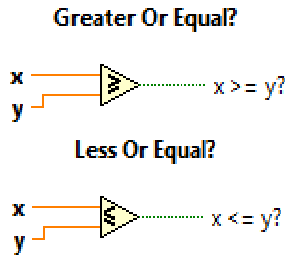
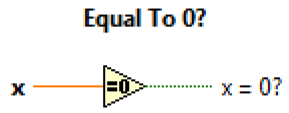
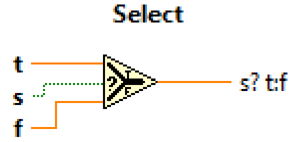
Figure 8 Block diagram for the Example 2

Comparison operators

Comparison operators are operators that allow verification of whether input values are greater, smaller or equal to some other values. The result of the comparison operator is usually the logical value (true or false) except for the ternary operator (Select). The basic comparison operators are listed in the Table 5.

Table 5 Comparison operators

Operator	Functionality
 	Checks whether inputs are equal / different.

	Checks whether the value of the upper input is higher / lower than the value at the bottom input.
 Greater Or Equal? x y x >= y? Less Or Equal? x y x <= y?	Checks whether the upper input is greater than or equal to / less than or equal to the value at the bottom input.
	Checks whether the input is equal to 0. Note: there are blocks for checking the input signal: - different from zero - smaller / smaller or equal to zero -higher / greater or equal to zero
	Select operator. Equivalent to the ternary operator of other programming languages. Based on the boolean value “s” at the entrance to the output, the value of the output t or f is transmitted. If true to output is forwarded to the input at t, otherwise the output is sent to the input f.

Example 3.

Virtual Instrument needs to be created which will sum up two numbers (A and B) if the switch on the front panel is turned on (true state) and subtract two input numbers if the switch is turned off (false state). If the solution is greater than 0 turn on the LED indicator on the front panel.

Designing the front panel

In order to provide the user with the ability to enter two numbers, two numeric controls (A and B) must be placed on the front panel. To select a operation, programer needs to position one switch on the front panel. Also, it is necessary to place one LED indicator that will indicate whether the solution is positive or not.

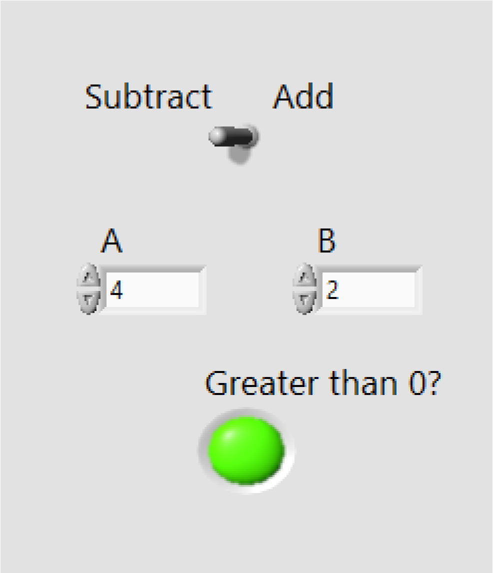


Figure 9 Possible front panel for the Example 3

Adding the functionality

Wanted functionality can be accomplished using the Select operator. Select operator will send to its output the sum or the subtraction of values A and B based on the boolean value the switch sends on its output terminal.

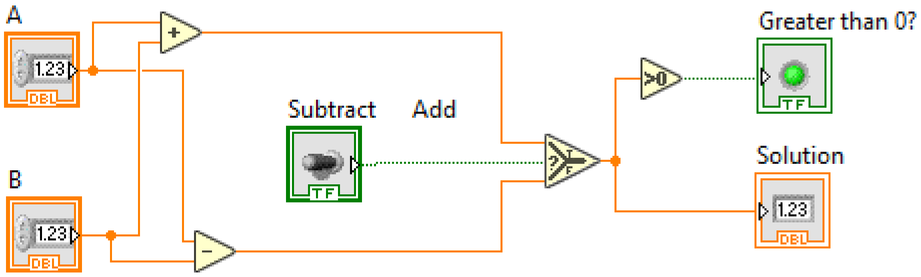


Figure 10 Block diagram for the Example 3

Assignment 2

Create a virtual instrument that takes on 6 input parameters that represent the vertices of a triangle in the Cartesian coordinate system (x1, y1, x2, y2, x3, y3). VI should calculate the distances between each of the vertices and show the solutions in adequate indicators ont the front panel.

Vi should also calculate the circumference of the triangle. If the circumference is smaller than 10 then a LED indicator on the front panel should light up.

5. Flow control

Programing in a certain textual programing language like C, C++, Java or C#, or using a graphical oriented environment, like LabVIEW, can be described as implementing one or several algorithms. An algorithm is a defined series of instructions, which, carried out, bring about a certain goal or result. The instructions often need to take on different modes depending on some input or calculated information, or need to be repeated any number of times. This modifies the sequence or the flow of the instructional set applied in a certain case.

LabVIEW makes this possible using entities named structures. In the *Functions* palette inside the *Programming* menu is a submenu *Structures*. *Variables* and *Decorations* are also placed in the same menu, but since they are not used to control the flow of the algorithm they will not be covered in this text.

Besides the wire connections (which define the sequence of connected nodes) one of the more common methods of flow control is using the comparison nodes located in the *Functions* palette. However, in more complex decision trees a structure is usually a better choice. Some of the commonly used structures are:

- Case structure,
- Event structure,
- Timed structure,
- Conditional Disable structure,
- Sequence/Stacked structure.

They are used to define several different algorithm flows based on a certain condition.

Case Structure

A *Case Structure* is used to define alternative algorithm cases depending on an input value. They have an input that is used to define the condition for algorithm alternative cases selection. It is called a *Case Selector*. The alternative cases of the structure are automatically adapted to the data type connected to the *Case Selector*. Any data type is accepted (*Boolean, Integer, Double, String, etc.*), and the number of alternative cases is not limited (except by the possible values for a certain data type). Alternative cases are represented by individual frames that can be accessed via drop-down menu at the top of the structure. Cases can be added or removed through a right-click menu on the boundaries of the structure. The code contained in the frame of a certain case is executed if the value of the *Case Selector* corresponds to the values indicated in the case title (drop-down list entry). All the possible values of the input to the *Case Selector* need to be covered (have a case/frame that is executed for the value). This can be overcome by specifying a *Default* case for all those that don't need a specific piece of code.

The following example depicted in Figure 1 illustrates a simple use of the *Case Structure*. The code includes three controls (*Input1, Input2* and *SUM*) and an indicator on the user interface (*Front Panel*). The program includes two alternative algorithms depending on the *SUM* switch value. If the switch is pressed (value is *True*) the executed code provides the sum of the inputs as the result. However, if the switch is depressed the result displayed is the difference of the input values. The alternative cases are indicated by the frame top case title, which doubles as the *Case Selector* value that triggers the case execution. The points at which wires (and values) enter or exit the structure are called tunnels. For the exit tunnels it is important to have a value defined for every single case, otherwise the code can not be executed. If there is a case (frame) that does not have a value (wire) connected to an exit tunnel this is indicated by the empty (white) tunnel square on the boundary of the structure. The *Use Default if Unwired* option can be activated through the right-click menu. This structure is essentially equivalent to the *Switch-Case* command in other programming languages.

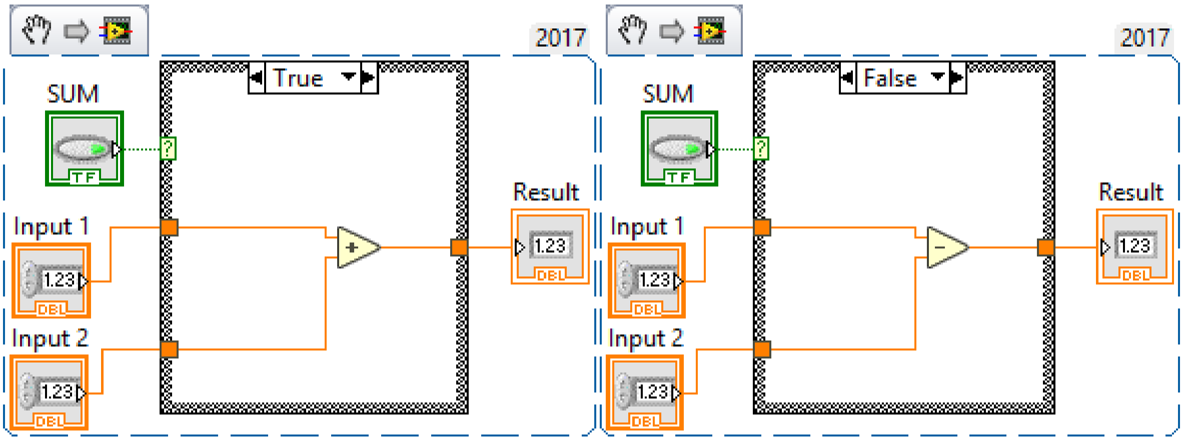


Figure 11 Example of using case structure.

Assignment 3

Produce a VI similar to the illustrated example with an arbitrary *Front Panel* design. The VI should be able to deliver a result based on the *Numeric Slider* selection of an arithmetic operation. The options should include summation, retraction, multiplication or division of two input values. The result should be calculated only for positive input values, otherwise the result should display 0.

6. Loops

As it was mentioned earlier, sometimes it is necessary to execute the same sequence of instructions (algorithm) a number of times consecutively. In order to avoid copying whole segments of code (or parts of *Block Diagram*) we use loops. As with most other programming languages, LabVIEW implements two types of loops:

- *For* loop.
- *While* loop.

The *For* loop is used when the needed number of code repetitions is available in advance. A good example would be to ask the user to input twenty numerical values which are then used to perform some calculations. The *While* loop is used when the number of repetitions is not available in advance, and is dependent on a certain condition or an action performed by the user. An example would be to ask the user to input as many numerical values as he would like, and to signal the end of input by typing in a negative value.

In LabVIEW the loops are structures marked by the loop frame. Any code within that frame is performed repetitively until the given condition is met.

For loop

The graphical representation of an empty For loop is illustrated in Figure 12.**Error! Reference source not found.**In the top left corner is the *Loop Count* value. This is the value that determines the number of times that the loop will be executed. Inside the loop frame is a blue indicator marked ***i***. This is the indicator of the current (when executing the program) loop iteration. It basically counts the loop executions, and is updated at the end of each iteration. The ***i < N*** condition is checked at the start of every iteration. If ***N = 0*** the code within the loop is not executed, not even once (since ***i*** starts at 0). At the end of loop execution the iteration counter states the value for the last fully executed iteration (normally it should be *N-1*).

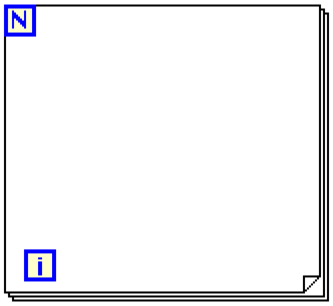


Figure 12 For loop appearance

While loop

An empty *While* loop is illustrated in Figure 13. In contrast to the For loop it does not contain a *Loop Count,* however, it is equipped with a terminal for conditional loop stopping. The condition is checked at the end of each iteration and if the logical value of the condition is *True* the execution of the loop is stopped (*Stop if True*). The iteration count ***i*** behaves exactly the same as within a For loop. An alternative variant of the loop has a *Continue if True* conditional terminal, that behaves as a negation of the described type. This corresponds to the *do-while* command in other programming languages. This option is available through a right-click on the terminal.

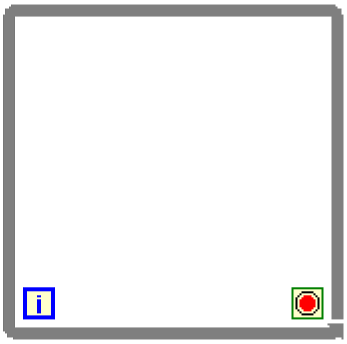


Figure 13 While loop appearance

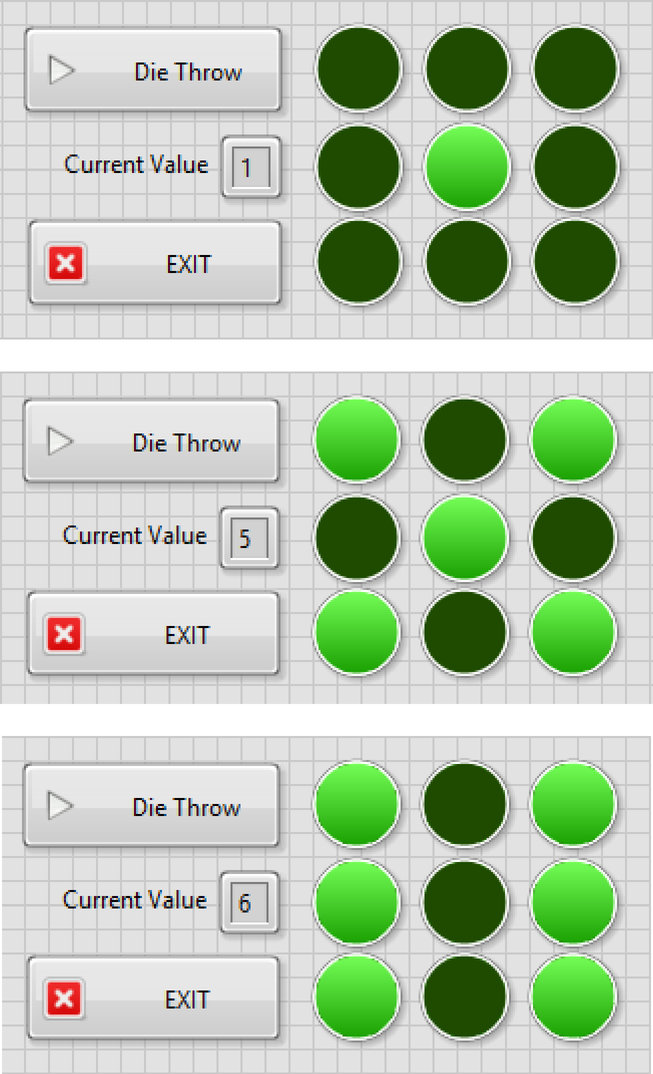
Assignment 4

Modify the assignment 2 so that the program runs until user presses a STOP button on the *Front Panel.*

Tip: Use the loop structures.

Assignment 5

Produce a VI that simulates die throws. The die is “thrown” when the button *Throw* is pressed. The VI should output a random integer value in the 1 – 6 range. The result should be displayed in the numerical indicator *Current Value* and with the help of a graphical 3×3 LED indicator as illustrated in Figure 14. The VI should run until the *EXIT* button is pressed.



Current Value 1

Current Value 5

Current Value 6

Figure 14 Example for die value 1 , 5 and 6 respectively

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

7. Data Structures

Data structures most commonly used in LabVIEW are *Arrays* and *Clusters.* These structures make the manipulation of large amounts of data easier and reduce the number of wires in the *Block Diagram.*

Arrays

Arrays, as with the command line programming languages, are used to store data of a same type (e.g. *I8, U32, Boolean, String,* etc.). Arrays can be explained as ordered sequences of data (values) of a certain type and length. If it is, for some reason, required to temporarily store and process a large amount of data of the same type, placing it in an array makes it easier to access individual values and perform calculations with them. Additional advantage comes in the form of an abundant palette of array specific tools that are used to this end. Without the use of arrays every single value (of some type) would have to be represented with it's own control or indicator. This would make reuse of the same code (within a loop) extremely difficult, as you would need to reference every single control/indicator individually. When using arrays only one reference is needed, and the individual values are addressed by numerical integer values – indices. This (index addressing) can therefore be manipulated mathematically, which is substantially simpler.

Arrays are especially convenient to store hardware data acquisition results, since these results are inherently of the same type (for one acquisition process). In LabVIEW arrays can grow dynamically as new values pour in and it is not needed to worry about memory reservation as in for instance C programming language. Every array value has an index assigned and this is essentially the address (position) within an array. Indices start with *0* and end with *N-1* for an *N* element array.

Arrays are added to the *Front Panel* as empty shells from the *Arrays, Matrix & Cluster* menu in the main palette. Each array can be either an *Indicator* or a *Control*, depending on the nature of the terminal inserted into the array shell. Data type is also tied in with the data type of the inserted *Indicator/Control.* The number of values displayed can be modified by "stretching" the outer limits of the array shell.

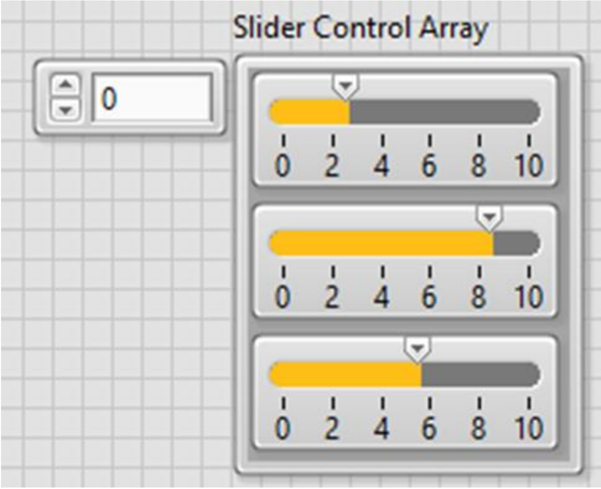


Figure 15 Slider Control Array

In Figure 15 an one dimensional (1D) array of slider controls is displayed. Three values are available simultaneously. This, however, does not indicate that the size of the array (number of elements) is equal to three, it only guarantees that it is not less than three.

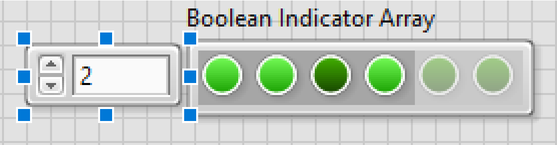


Figure 16 Boolean Indicator Array

In Figure 16 a 1Darray of Boolean indicators is displayed. The highlighted index selector is not set to the first element of the array (index 0), but to the third element (index 2). Notice that values at indices 2 through 5 have values [T, T, F, T]. The size of the array in this example is 6 (last index 5). The following values are not available – they are greyed out since they are not defined.

It is also good to note that in both figures 1D arrays are displayed. However, in Figure 15 it is displayed as a column, and in Figure 16 it is displayed as a row. This makes no difference whatsoever when dealing with 1D arrays, and is up to the programmer/user.

The default number of dimensions for arrays is one, but this can be extended if needed to any number of higher dimensions. A helpful visualization is to think of 2D arrays as tables, where one of the dimensions corresponds to row number and the other to the column number. A further expansion of this approach can be refined to represent the third dimension as the page number (of a book) on which the table is set, the fourth as the book volume (in a series of books), etc.

A 2D array is created thorugh a 1D array shell right-click menu option *Ad d dimension.* A 2D array of Boolean indicators is illustrated in Figure 8.

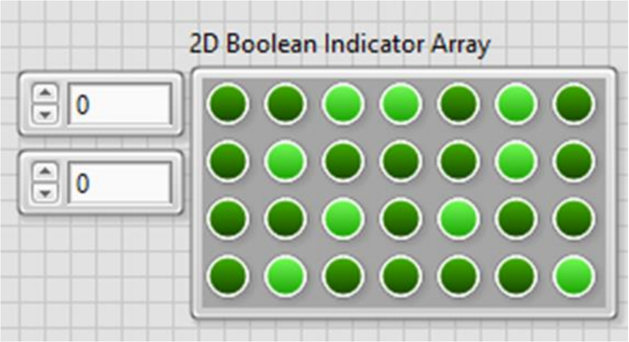
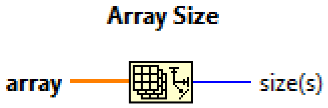
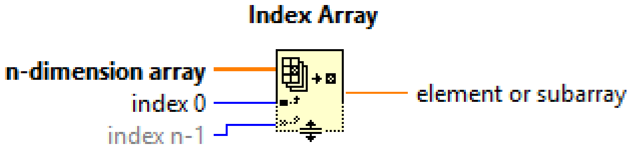
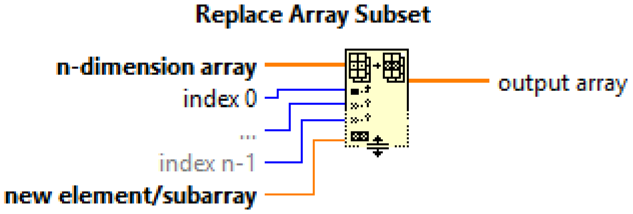


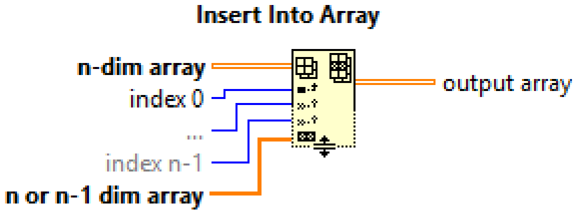
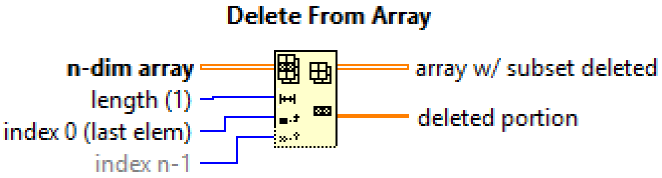
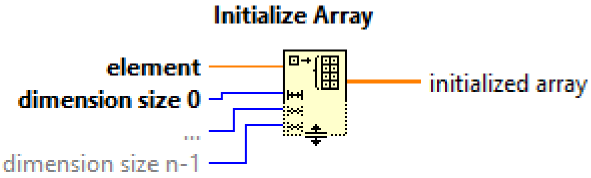
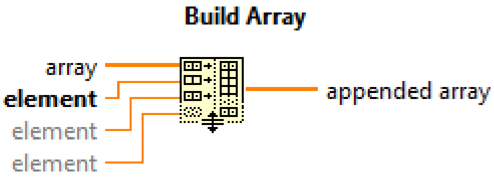
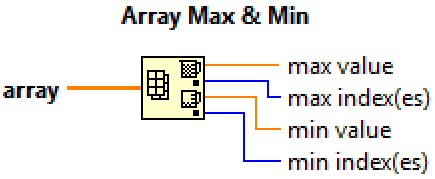
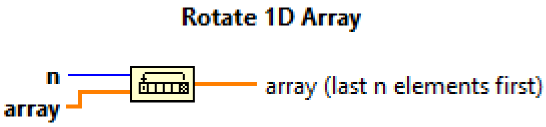
Figure 17 2D Boolean Indicator Array

Array palette tools

Table 6 gives an overview of some basic array tools available from the palette.

Table 6 Basic Array Operations

Tool description	Tool representation
Returns the number of elements in each of the **array** dimensions	
Returns the element or subarray of the n-dimensional array at specified **index**	 Index Array n-dimension array index 0 index n-1 element or subarray
Replaces the element or subarray of the **n-dimensional array** at specified **index** with the **new element/subarray** (number of elements of the original array does not change)	 Replace Array Subset n-dimension array index 0 ... index n-1 new element/subarray output array

Inserts the element or subarray (**n or n-1 dim array**) into the **n-dimensional array** at specified **index** (number of elements of the original array increases)	 Insert Into Array n-dim array index 0 ... index n-1 n or n-1 dim array output array
Deletes an element or subarray from the **n-dimensional array** of **length** elements starting at **index**. Both the new (reduced array) and the deleted portion are available as outputs	 Delete From Array n-dim array length (1) index 0 (last elem) index n-1 array w/ subset deleted deleted portion
Creates an n-dimensional array in which every element (a total of **dimension size**) is set to the value of **element**	 Initialize Array element dimension size 0 ... dimension size n-1 initialized array
Concatenates (joins together) same dimension arrays/elements, or appends (adds to the end) an (n-1)-dimensional **element** to the **array**	 Build Array array element element element appended array
Returns the maximum and minimum values found in the **array**, along with their indices	 Array Max & Min array max value max index(es) min value min index(es)
Moves the elements of the **array** by **n** positions in the direction of the sign (+ from left to right; - from right to left). The number of elements remains the same overflow moves to the opposite end	 Rotate 1D Array n array array (last n elements first)

An example of an array manipulation VI Block Diagram is illustrated in Figure 18, and the result is displayed in Figure 19.

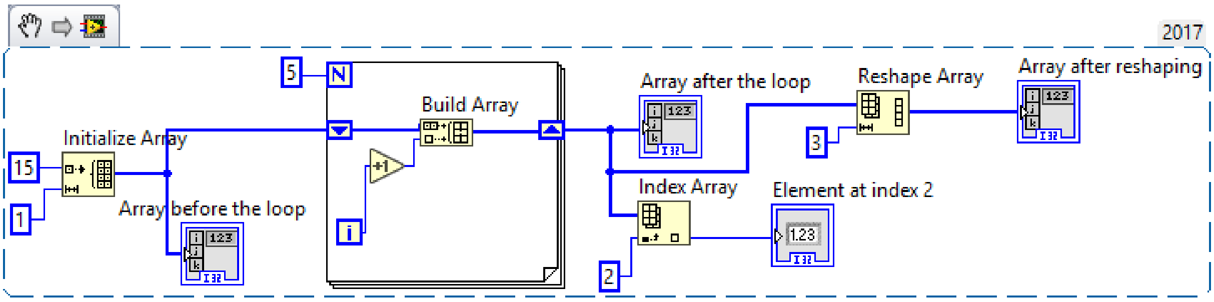


Figure 18 Example of an array manipulation

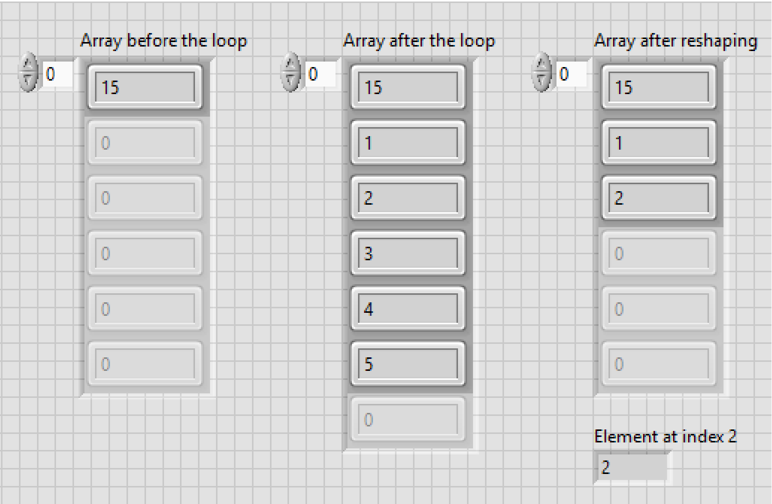


Figure 19 Example array operations execution result

Loops and arrays (auto - indexing)

To access consecutive single values of an array placed outside the loop from inside the loop the simplest approach is to use the auto-indexing option on the tunnel indicator. This is the default option when dealing with *For* loops, and can be activated manually (right-click menu *Tunnel mode-indexing*) when using a *While* loop. It can be deactivated in a similar fashion on a *For* loop tunnel. This tunnel mode strips a dimension from the data coming into the loop, and produces the same result as *Index Array* from **Error! Reference source not found.** would produce with iteration counter ***i*** value connected to the **index** input.

When crossing through an indexing tunnel from inside the loop to the outside a dimension is added to the data connected to the tunnel. The result is like the one *Build Array* from **Error! Reference source not found.** would produce if values (of the wire connected to the tunnel) for every single iteration of the loop were appended through this operation to the previous result.

An example of a loop with indexing mode of the tunnel enabled is displayed in Figure 20.

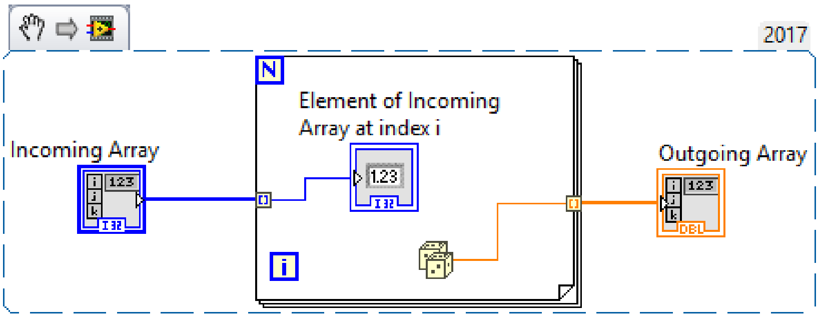


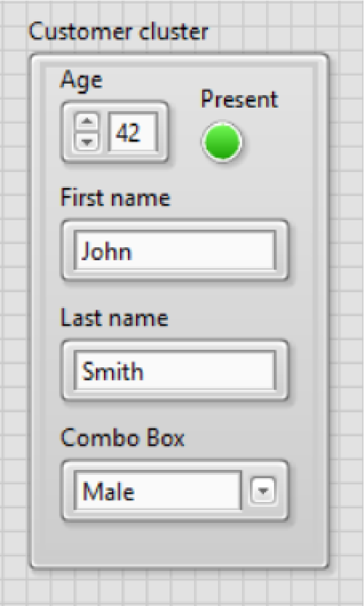
Figure 20 Indexing tunnel mode

It is important to note that the *Loop Count* (*N*) does not have to be connected. The number of elements of the *Incoming Array* limits the number of iterations. It is possible to lower the number of iterations (compared to *Incoming Array* size) by connecting a smaller value to *Loop Count* (*N*), but it is not possible to increase it. The *Outgoing Array* will contain random values that were generated in each of the loop execution iterations. The size of the *Outgoing Array* will be the same as that of the *Incoming Array* in this example.

Clusters

Clusters are a data structure that can contain multiple types of data in one package. By their nature they correspond to *structures* in the C programming language. In difference to arrays, which can change in size dynamically, clusters have a constant size. They are formed by grouping different (or same) types of data. Individual elements of clusters are accessed by unpacking or *unbundling* a cluster and wiring the appropriate connection terminal of the *Unbundle* node. The creation of the cluster is very similar to creating arrays. From the *Array, Matrix & Cluster* menu an empty cluster shell is dropped onto the workspace. It is then populated with indicators/controls of the data types that you wish to use. In difference to arrays, several indicators/controls can be dropped into a cluster and they can all be of different data types (but don’t have to be). A single cluster is either an indicator or a control, and can therefore only be populated with terminals of the corresponding type. In Figure 21**Error! Reference source not found.** a cluster containing several different controls is displayed. It consists of one I32 numeric, one Boolean, two string and a combo box control.

As soon as a terminal is created on the *Front Panel*, a representation of it also appears in the *Block Diagram*, as is the case with all terminals (no matter what type) in LabVIEW.



Customer cluster

Age: 42

Present

First name: John

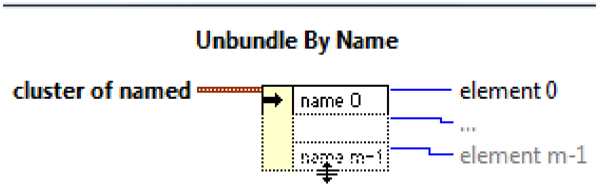
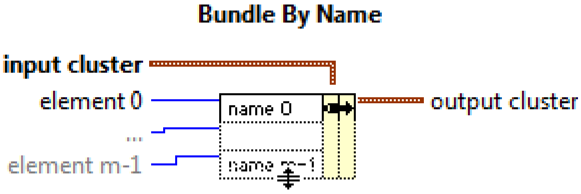
Last name: Smith

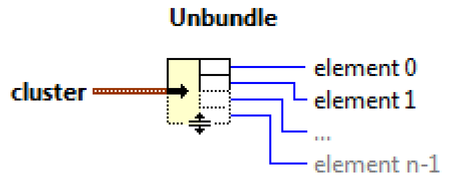
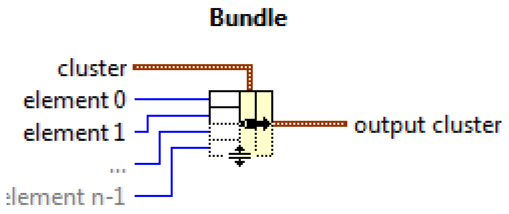
Combo Box: Male

Figure 21 An example cluster with different customer information

As with arrays, clusters have a separate palette that contains many specific tools useful for their manipulation. Basic cluster manipulation tools are represente in the Table 7.

Table 7 Basic cluster manipulation tools

Tool description	Tool representation
Unpacking individual (or multiple) elements of the cluster package by designated name selection.	 Unbundle By Name cluster of named name 0 name m-1 element 0 ... element m-1
Overwrites the designated element of the **input cluster**	 Bundle By Name input cluster element 0 ... element m-1 name 0 name m-1 output cluster

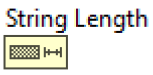
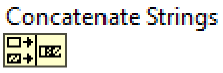
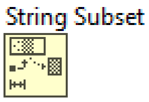
Unpacks all the elements of the input **cluster**	 Unbundle cluster element 0 element 1 ... element n-1
Builds a cluster of elements wired to the input terminals. If a cluster is wired to the definition input (upper terminal) than the node can only replace existing cluster elements	 Bundle cluster element 0 element 1 ... :lement n-1 output cluster

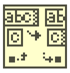
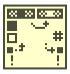
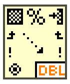
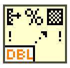
String functions

Strings maybe do not belong to this chapter but their manipulation is very closely related to the manipulation of arrays so they are included in here.

As mentioned earlier Strings are basically streams of characters – a text. Text manipulation is very often needed in practice. For example – to retrieve a single word from the whole sentence or to compose a text report from a Virtual Instrument. Some of the useful functions that the programmer can use with the Strings are represented in the Table 8.

Table 8 Useful String functions.

Tool description	Tool representation
	Calculates the number of characters in the string.
	Joins multiple strings together. Can be extended for multiple inputs.
	Returns a substring of the input string based on the beginning index and the length parameter.

Search/Split String 	Divides an input string into two strings based on the second input string.
Search and Replace String 	Finds and replaces one input string within another input string. Can replace all occurrences or just the first.
Scan From String 	Converts the input string to number based on the format input. Uses C language based string formatting.
Format Into String 	Converts the input number to string based on the format input. Uses C language based string formatting.

Assignment 6

Create a virtual instrument that enables the user to enter numbers through the numerical control and a button on the front panel. Each number should be stored in the array of numbers. Data entry is over when user enters -1 in the numerical control.

After the data entry Vi calculate the arithmetic mean of all entered numbers (except -1) and what was the maximum value, minimum value entered.

Assignment 7

Create a virtual instrument that enables user to enter values about different fish into a cluster on the front panel.

Data to be entered about a fish are: name (String), weight (double), length (double), is it a seafish (boolean). User can enter values for 20 fishes. When the data entry is over the program should display the average weight af all seafish.

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

8. Use of acquisition devices (NI MyDAQ)

As it was previously mentioned, one of the main characteristics of the LabVIEW programming environment is data processing and analysis. However, in order to work with data, it first needs to be acquired and imported into the environment. This is greatly simplified using peripheral data acquisition devices produced by the same company that produces LabVIEW – National Instruments.

One of these devices is called MyDAQ. It is a relatively low cost device designed for student and educational applications. There are several options for data acquisition available within it. It is equipped with A/D and D/A convertors so that analog data can be either input into the device, or output from it and there are also digital input/output terminals.

It is connected to the computer via an USB port and with the appropriate driver support (optional installation with the LabVIEW environment package) it is very easy and quick to setup and be put to work acquiring (or delivering) data.

NI ELVISmx Instrument Launcher

This software is a program tool that can use MyDAQ as a peripheral device to acquire or deliver analog and digital signals. No LabVIEW programming knowledge is needed in order to use it. The user interface is graphical and mimics the look of standard stand-alone instruments typical for electrical measurements and testing. It can also be used in combination with other NI equipment (e.g. ELVIS after which it was named and for which it was primarily developed).

Available modules are stated in the next list. Detailed instructions are not considered in this material.

- Digital Multimeter
- Oscilloscope
- Function Generator
- Variable Power Supply
- Bode Analyzer
- Dynamic Signal Analyzer
- Arbitrary Waveform Generator
- Digital Writer
- Digital Reader
- Impedance Analyzer

Not all modules are available to use with the MyDAQ device. For *Digital Multimeter* applications there is an extra set of probes in the package (separate terminals).

Data acquisition and MyDAQ

When a sensor detects some physical property (e.g. temperature, pressure, illumination, etc.) it actually converts the property value into an electric signal and sends this signal to a processing device for further analysis. The receiving and primary processing of the signal, on its way to the computer, is actually done by data acquisition devices or DAQs. The computer (or the program running on it) needs to know where the required data comes from, how often does it need to “read” the value and how long does it need to keep “reading” it. When dealing with analog signals an additional step of analog-digital signal conversion is needed. This is done on order to reduce the data to a manageable size, since an analog signal contains an infinite amount of data (theoretically).

When acquiring data a part of the acquired values is correlated to the measured property, but there is also a second part of data which does not describe the measured property. This is called noise. Noise covers all the unwanted occurrences that have found their way into our measurement data and have influenced the values that we have obtained in the process of measurement. There are different types of noise and they can stem from the measurement system itself, imperfections of the sensor element, temperature fluctuations, random background electromagnetic signals etc. If the level of noise is much lower than the level of the signal that we are trying to measure, it can sometimes be ignored, but when this is not the case there are a couple of things that can be done to improve the situation. The noise frequency is often higher than the measured signals, and this can produce some difficulties when digitizing the resultant total signal. Depending on the source of the noise some of the possible steps are using higher quality detectors, increasing the number of measurements and averaging the results, filtering (hardware) the signal prior to acquisition (before the DAQ device), filtering (software) the signal after the acquisition (after the DAQ device).

When the noise originates from the environment in which the measurements are made, a higher quality sensor cannot correct the problem significantly. An increased number of measurements and then averaging or maybe signal filtering are the best approach. Filtering prior to ADC can be costly, since it includes additional hardware but can help with aliasing frequency problems due to higher (than measured signal) frequency conversions. Filtering post ADC assumes the use of digital filters, but also of higher sampling frequency (oversampling) in order to avoid aliasing problems.

The signal from the sensor needs to be conditioned for the acquisition device. The DAQs have a working range (voltage) and the incoming signals often need to be amplified and/or shifted to fit into the declared DAQ range. Filtering is also a frequent demand, but it has already been discussed in the previous paragraph.

MyDAQ is a device which offers analog signal connection, AD conversion and forwarding acquired data to the LabVIEW environment.

Besides the analog inputs there is also a series of digital connections which can be either incoming or outgoing. Digital inputs, depending on the standard and type (CMOS, TTL) are mostly designed for two separate voltage ranges for a logical 0 (*False*) and a logical 1 (*True*).

When performing AD conversion an important parameter to have in mind is the convertor resolution, which is defined by the number of bits available for coding the values of the input signal, as well as the dynamic voltage range the convertor is designed for. By increasing the number of coding bits the number of available voltage levels increases (with smaller steps between neighbor levels). This means that the digitized sample value will be closer to the real value of the input analog signal. However, due to smaller steps between neighboring digitized signal levels the noise sensitivity increases.

When acquiring signals from LabVIEW it is necessary to define:

- Device to be used for acquisition (sometimes multiple devices are available)
- The correct physical input terminal
- The sampling frequency

The complete set of acquisition instructions is called a *Task*. Data acquisition is possible even outside of the LabVIEW programming environment with the use of *Measurement and Automation Explorer (MAX)* tool. This approach is useful when testing the functionality of data acquisition. From inside the LabVIEW environment the easiest way to setup acquisition is through *DAQ Assistant Express VI*. It is located in the *Measurement I/O palette*, under *DAQmx-Data Acquisition Group.*

When the appropriate node is dropped onto the *Block Diagram* a user dialogue is started which guides the user through the *task* creation. First step is to choose if the signals are to be acquired (input) or generated (output). Next you choose the nature of the signals – analog or digital. On the analog branch there are a number of options available, but most applications deal with voltage. After this has been set a new list of available analog voltage connections is displayed, and the correct one needs to be selected. The final screen offers the possibility of testing the configured data acquisition (*Run* on the toolbar at the top with numerical/table or graphical representation of values directly beneath it), and some fine tuning of timing options (sampling frequency) as well as the dynamics of the signal (voltage range). The timing options include:

- Acquisition mode (one sample, multiple samples, continuous acquisition)
- Samples to read (with multiple samples)
- Rate (sampling frequency selection)

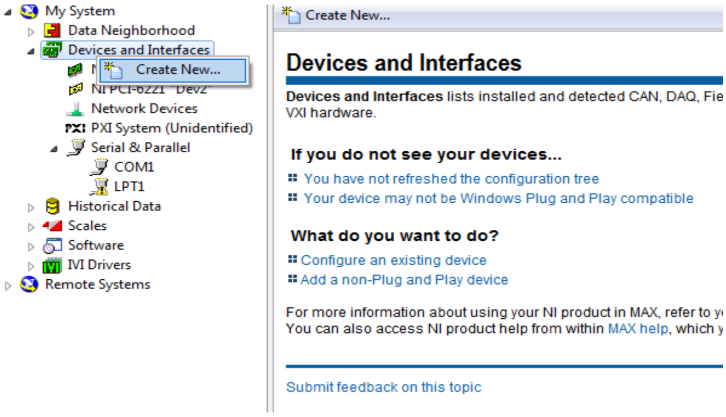
After clicking *OK* the configured task is created and prepared for use within the VI. The acquired data is available from the *Data* terminal, and is usually of the *Dynamic Data Type* (*DDT*).

Assignment 8

Using LabVIEW programming environment construct a software multiband audio signal equalizer for signals acquired from the audio input of the MyDAQ. The resulting signal should be routed to the audio output of the device.

9. Simulating an acquisition device

In case a user does not have a DAQ available it is possible to simulate one using the DAQmx program tools. Before it can be used it needs to be created in the *MAX*. Right-clicking the *Devices and Interfaces* section opens a menu with the *Create New* option as in Figure 22.



Devices and Interfaces

Devices and Interfaces lists installed and detected CAN, DAQ, Fie
VXI hardware.

If you do not see your devices...

- You have not refreshed the configuration tree
- Your device may not be Windows Plug and Play compatible

What do you want to do?

- Configure an existing device
- Add a non-Plug and Play device

For more information about using your NI product in MAX, refer to y
You can also access NI product help from within MAX help, which y

Submit feedback on this topic

Figure 22 NI MAX new device

In the user dialogue the *Simulated NI-DAQmx Device or Modular Instrument* should be selected as in Figure 23.

In the next window (Figure 24) a device to be simulated needs to be picked from a list of all devices available for simulation. The devices differ in characteristics such as the number of analog and/or digital inputs/outputs.

After completing the procedure the selected device becomes available in the LabVIEW environment as a simulated acquisition device. The further use of the simulated device is identical to that of a real, physical device. The *DAQ Assistant* user interface behaves in the previously described way. The signals coming into the simulated device are, of course, also simulated, so it is required to select the desired signal from the options available.

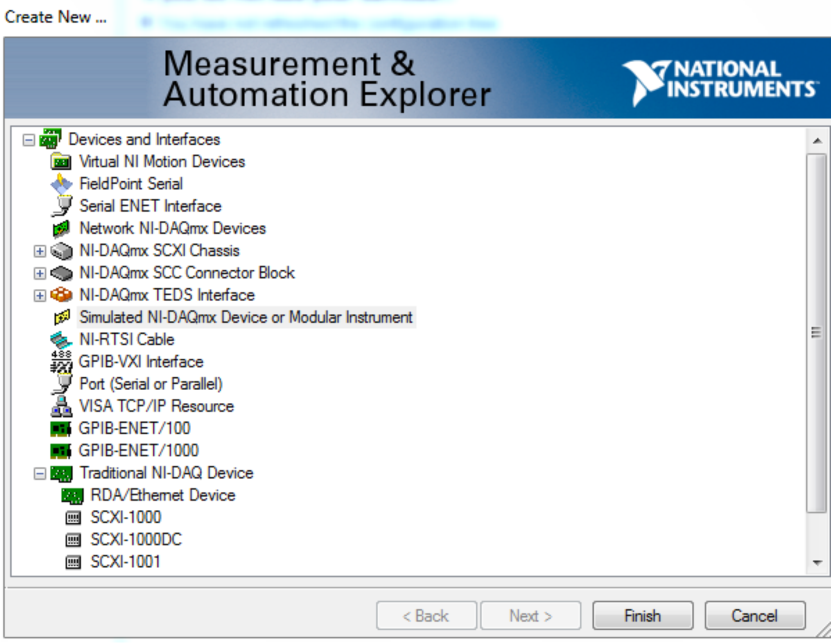
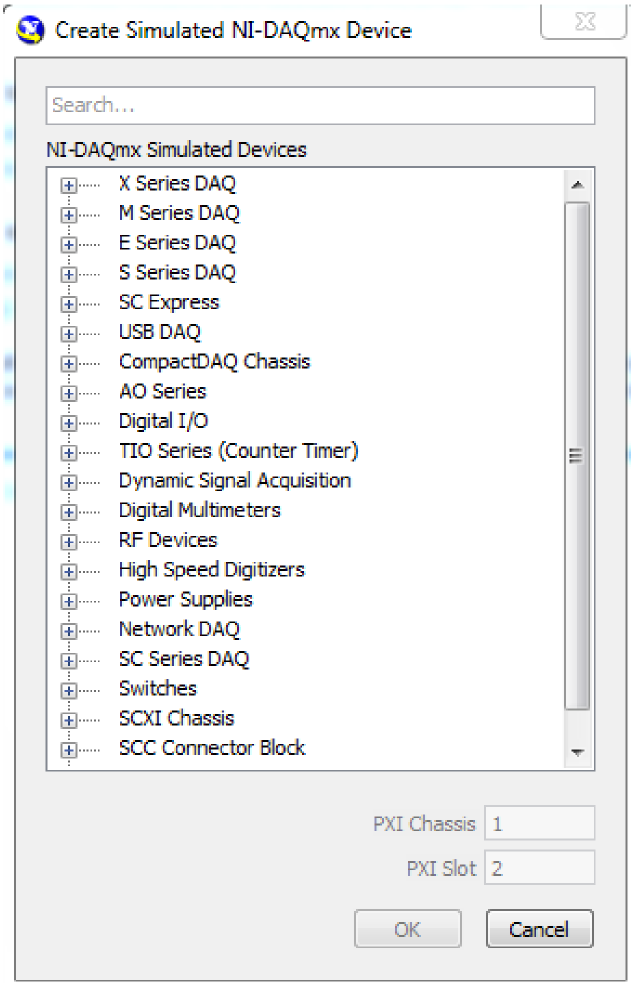


Figure 23 NI MAX simulate device



Create Simulated NI-DAQmx Device

Search...

NI-DAQmx Simulated Devices

- X Series DAQ
- M Series DAQ
- E Series DAQ
- S Series DAQ
- SC Express
- USB DAQ
- CompactDAQ Chassis
- AO Series
- Digital I/O
- TIO Series (Counter Timer)
- Dynamic Signal Acquisition
- Digital Multimeters
- RF Devices
- High Speed Digitizers
- Power Supplies
- Network DAQ
- SC Series DAQ
- Switches
- SCXI Chassis
- SCC Connector Block

PXI Chassis 1

PXI Slot 2

Figure 24 NI MAX list of devices available for simulation

10. Conclusion

LabVIEW is a tool for graphical programming designed for easy (or easier) physical environment measurement using different data acquisition devices.

Programs produced in LabVIEW are called *Virtual Instruments*. They are basically designed to imitate and improve the functionalities of stand-alone measurement devices. The programming environment consists of two main windows:

Front Panel

Adding controls and indicators to this window enables the user to input information or receive it from the virtual instrument. Representations of these terminals are automatically placed in the *Block Diagram*. The resulting user interface is graphical in nature with all forms of interactive objects such as buttons and switches, LEDs, graphs, etc.

Block Diagram

This window is the programmers view, and all the functionality of *Front Panel* elements is setup here. The most important paradigm of LabVIEW programming is *Dataflow*, and it determines the order in which sections of code are executed. The order is not dependent of the position but of the wire connections connecting code sections. Wires transport data between nodes. Nodes correspond to classical commands. The usual programming tools , such as case selectors, loops and similar are available in a graphic form.

Data structures are available and used to manipulate large amounts of same type data (arrays) or packaged data of different types (clusters).

Data acquisition devices are easy to setup in LabVIEW, and acquired data can easily be accessed and processed.

If a physical DAQ device is not available it can be simulated.

The primary focus of this instructional material was to serve as a supplement to the laboratory exercises that are part of the course *LabVIEW graphical programming* at the Zagreb Univesity of Applied Sciences. The scope of the material was not aimed beyond that of a quick reference guide. For a more detailed explanation of concepts and functionalities an interested user is directed to try out the following books:

- Jeffrey Travis, Jim Kring: LabVIEW For Everyone; 3rd edition, Prentice Hall 2006; ISBN-13: 978-0131856721
- Ronald W. Larsen: LabVIEW For Engineers; Prentice Hall 2011; ISBN-13: 978-0136094296

The National Instruments webpages also present an abundance of materials available in the form of presentations or videos, and should provide a valuable additional resource on specific subjects, if not as a general overview.

List of figures

List of tables

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.