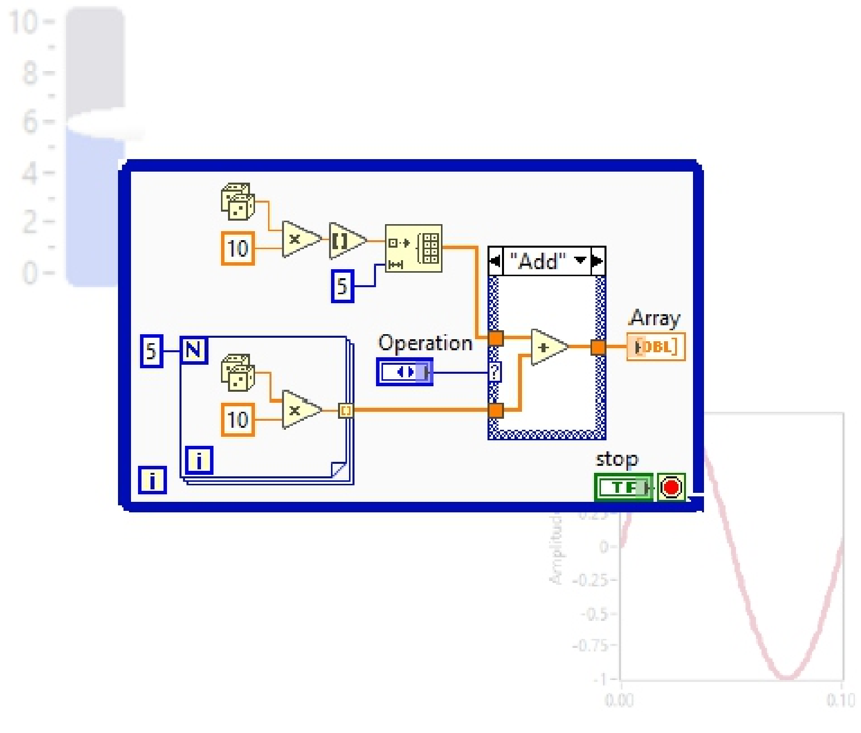
Alexandra FODOR Ionel Horea BACIU

VIRTUAL INSTRUMENTATION

- applications -



U.T.PRESS
Cluj-Napoca, 2025
ISBN 978-606-737-768-2

Alexandra FODOR **Ionel Horea BACIU**

VIRTUAL INSTRUMENTATION

-applications -

U.T.PRESS
Cluj - Napoca, 2025
ISBN 978-606-737-768-2

Editura U.T.PRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Recenzia: Conf.dr.ing. Gabriel Chindriș
Conf.dr.ing. Liviu Viman

Pregătire format electronic on-line: Gabriela Groza

ISBN 978-606-737-768-2

Contents

Preface

This book represents an applied handbook for the Virtual Instrumentation laboratory, edited and held by the authors at the Faculty of Electronics, Telecommunications, and Information Technology. The handbook is meant for the fourth-year students of the Applied Electronics specialization but is appropriate also for anyone who wants to learn about graphical programming language.

The content is structured in 12 chapters, the applications being grouped according to the main functions surrounded by the code. The applications have been developed and refined through years of classroom experience and are closely aligned with the learning objectives of engineering programs in electronics and even automation. This handbook aims to build both competence and confidence in using graphical programming language as a tool for modern engineering solutions.

Our most sincere thanks go to all colleagues, the professors who helped us in editing this handbook, especially to the reviewers who took part of their valuable time to share their opinions, and last but not least, to the students whose curiosity and feedback shaped the content of these pages.

Cluj-Napoca, 2025
The authors

1 LabVIEW Introduction

1.1 Definitions

LabVIEW is a graphical programming environment used for developing measurement, test, and control systems. The applications developed in LabVIEW are called virtual instruments, or VIs. These are composed of three basic elements: **Front Panel**, **Block Diagram** and the **icon.**

LabVIEW follows a **Dataflow** model for running Vis, which is based on the idea that modifying a variable implies recalculating the value for another variable dependent on the first one.

A virtual instrument can contain one or multiple subVIs. These can be structured as a project, observed below.

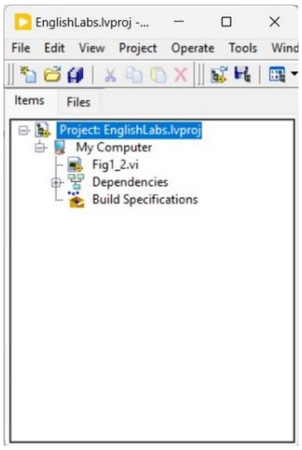


Figure 1.1 Project structure.

LabVIEW uses three types of files, one for its projects, ***.lvproj**, one for the VIs ***.vi** and one for the personalized control elements, ***.ctl**.

The folders that appear in the project structure do not necessarily appear as folders on disk, but rather subgroups of the used subVIs.

1.2 Front Panel

The Front Panel, like the interface of a physical instrument, represents the user interface, and can be seen in Figure 1.2.

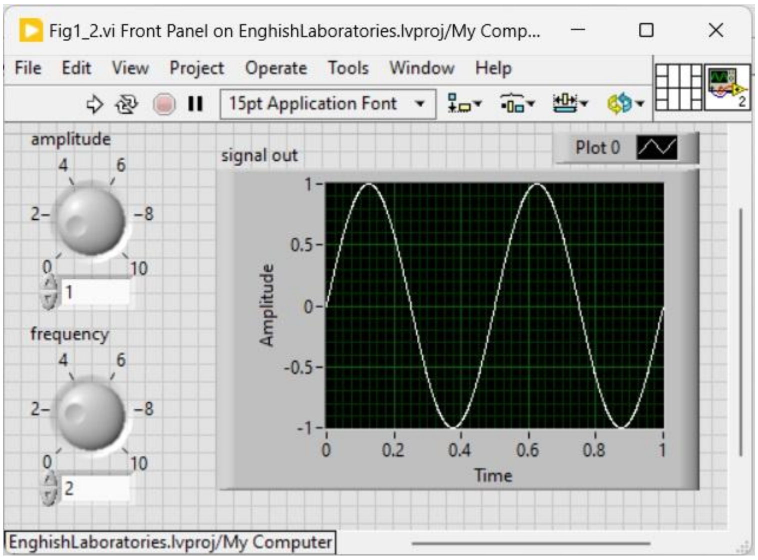


Fig1_2.vi Front Panel on EnghishLaboratories.lvproj/My Comp...
File Edit View Project Operate Tools Window Help
15pt Application Font
amplitude
frequency
signal out
Plot 0
Amplitude
Time
EnghishLaboratories.lvproj/My Computer

Figure 1.2 VI Front Panel.

The Front Panel is comprised of a combination of control and indicator elements. The control elements simulate input devices and give data to the Block Diagram of a VI. The indicators simulate output instruments used for displaying acquired or generated data from the VI diagram.

The controls and indicators can be selected and inserted in the Front Panel from the Control Functions Palette. This can be made visible through two methods: the first is by accessing the optional menu delivered by right clicking anywhere in the workspace of the Front Panel and the second is by selecting the main menu **View** -> **Controls Palette**.

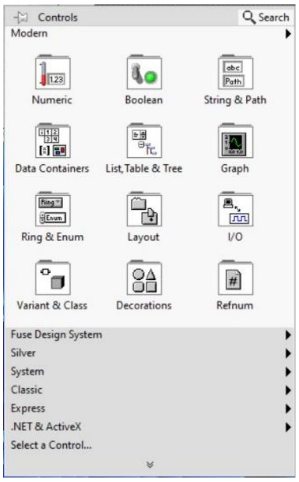


Figure 1.3 Controls and indicators palette.

The controls and indicators are grouped as follows: numerical elements, Boolean (logic) elements, arrays, matrix and cluster elements, list and table elements, graph elements, input/output elements, dialog elements, etc. These elements can be personalized by changing their shape or label, using tools instruments, presented in Figure 1.4. For making the Tools palette visible, select **View -> Tools Palette**. This allows the automated selection of an instrument. In case manual selection is needed, the available buttons are: a cursor for selecting and modifying position, a button for modifying elements text and labels, a button for selecting colors, etc.

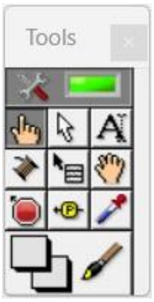


Figure 1.4 Tools Palette

The instruments palette has also control elements for the Block Diagram. These allow connecting elements through wiring, adding breakpoints for functions and structures, or viewing data on the wires.

1.3 Block Diagram

Each element on the Front Panel has a corresponding terminal in the Block Diagram. The **Block Diagram** is the environment where the algorithm is inserted, through graphical blocks for the virtual instrument to be created. An example of such an algorithm can be seen in Figure 1.5.

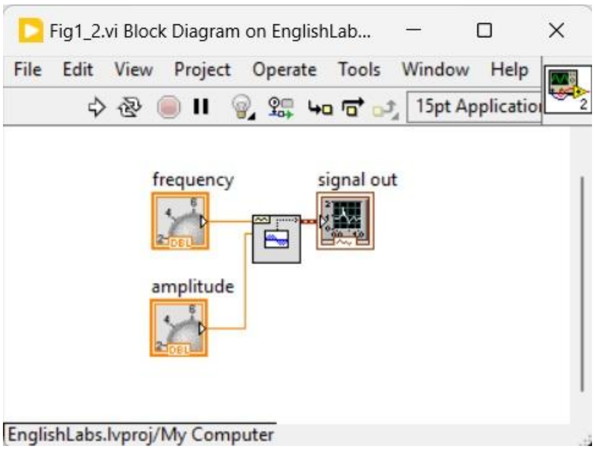


Figure 1.5 Block Diagram.

Except for the terminals, which are the connection between the Front Panel and Block Diagram, the latter can contain constants, nodes (which can be functions) subVIs and structures, connection wires and free labels for documenting code parts.

LabVIEW offers a wide range of functions. These can be accessed via two methods: Firstly, by right clicking anywhere in the Block Diagram, and secondly, by selecting from the main menu **View->Functions Palette.**

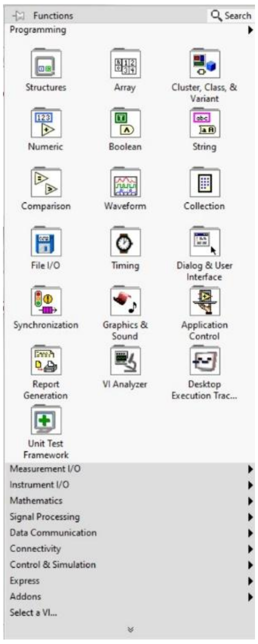


Figure 1.6 Functions Palette.

The functions palette contains structures, numeric functions, Boolean functions, array handling functions, comparison functions, waveforms functions, file handling functions, etc. LabVIEW allows searching for functions with known names.

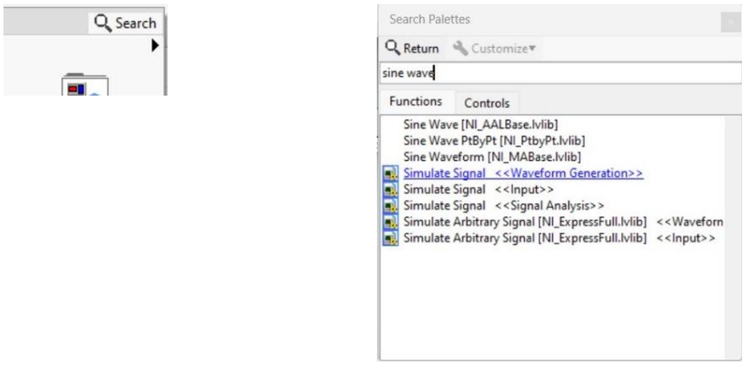

Search
Search Palettes
Return
Customize
sine wave
Functions
Controls
Sine Wave [NI_AALBase.lvlib]
Sine Wave PtByPt [NI_PtbyPt.lvlib]
Sine Waveform [NI_MABase.lvlib]
Simulate Signal <<Waveform Generation>>
Simulate Signal <<Input>>
Simulate Signal <<Signal Analysis>>
Simulate Arbitrary Signal [NI_ExpressFull.lvlib] <<Waveforn
Simulate Arbitrary Signal [NI_ExpressFull.lvlib] <<Input>>

Figure 1.7. Searching for functions.

LabVIEW also allows searching for already implemented examples for certain functions.

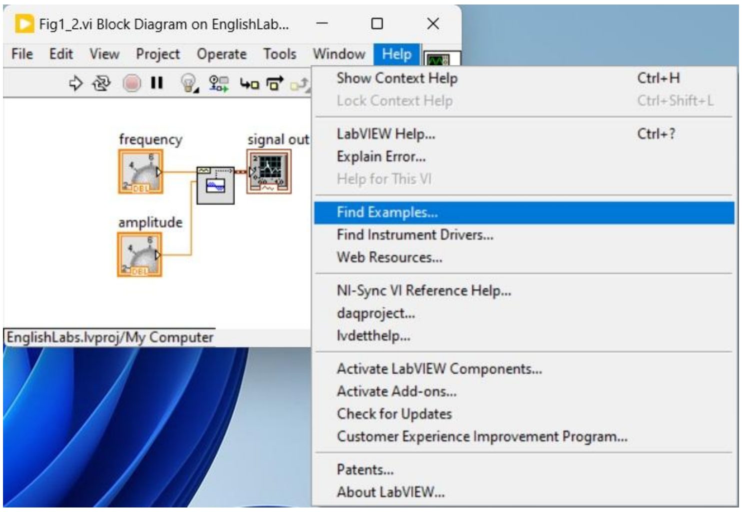
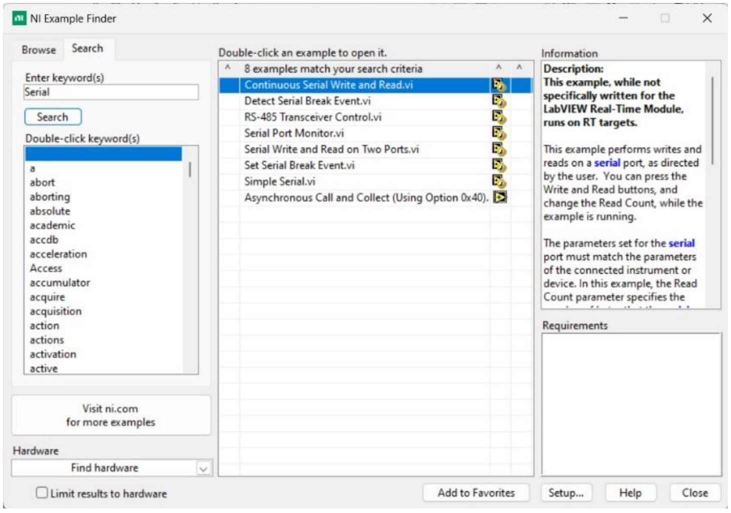


Fig1_2.vi Block Diagram on EnglishLab...
File Edit View Project Operate Tools Window Help
frequency
signal out
amplitude
EnglishLabs.lvproj/My Computer
Show Context Help Ctrl+H
Lock Context Help Ctrl+Shift+L
LabVIEW Help... Ctrl+?
Explain Error...
Help for This VI
Find Examples...
Find Instrument Drivers...
Web Resources...
NI-Sync VI Reference Help...
daqproject...
lvdetthelp...
Activate LabVIEW Components...
Activate Add-ons...
Check for Updates
Customer Experience Improvement Program...
Patents...
About LabVIEW...

Figure 1.8 Searching for examples.

The window below will be opened when selecting the „Find Examples" menu item.



8 examples match your search criteria

Continuous Serial Write and Read.vi
Detect Serial Break Event.vi
RS-485 Transceiver Control.vi
Serial Port Monitor.vi
Serial Write and Read on Two Ports.vi
Set Serial Break Event.vi
Simple Serial.vi
Asynchronous Call and Collect (Using Option 0x40).

Description:
This example, while not specifically written for the LabVIEW Real-Time Module, runs on RT targets.

This example performs writes and reads on a serial port, as directed by the user. You can press the Write and Read buttons, and change the Read Count, while the example is running.

The parameters set for the serial port must match the parameters of the connected instrument or device. In this example, the Read Count parameter specifies the

Figure 1.9 Examples.

In the Block Diagram, apart from the regular functions, there are also controls, indicators, and constant elements.

Control and constant elements are similar because they give certain information to the output. The difference between them relates to the fact that controls can be modified while a VI is running, and constants are not modifiable.

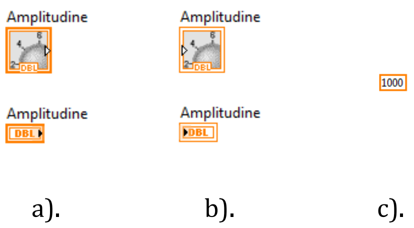


Figure 1.10 a) Control element; b) indicator element; c) constant.

The default appearance mode for a control or indicator element can be modified by right-clicking on an element and selecting the option **View As Icon**.

In the Block Diagram, different data types are represented with different colors.

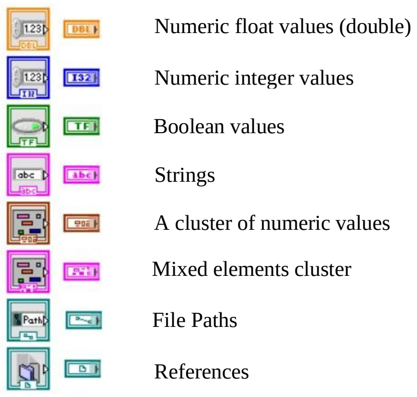


Figure 1.11 Data type representation.

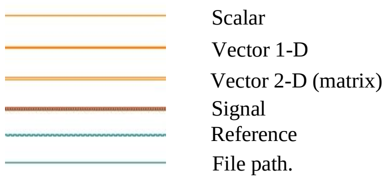


Figure 1.12 Wire data types.

1.4 Graphic representation (icon) for the virtual instrument

A VI's **icon** is the graphic image which represents a VI in another VI. It is recommended to be as explicit as possible, to reflect the function the VI has.

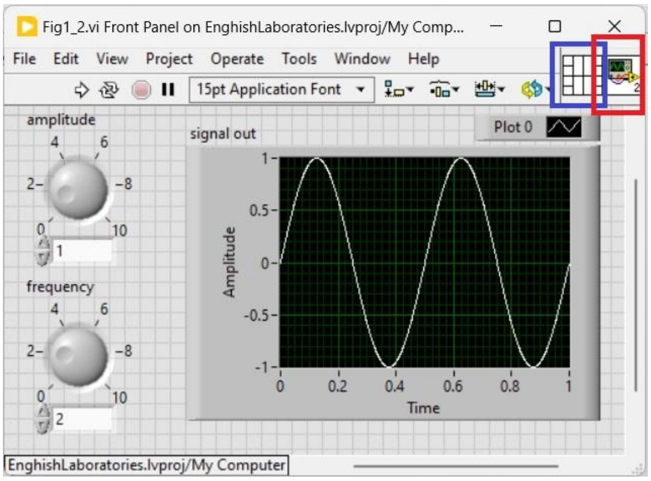
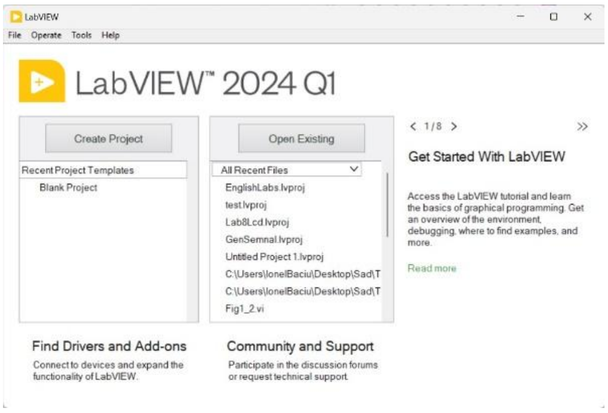


Fig1_2.vi Front Panel on EnghishLaboratories.lvproj/My Comp...
File Edit View Project Operate Tools Window Help
15pt Application Font
amplitude
frequency
signal out
Plot 0
Amplitude
Time
EnghishLaboratories.lvproj/My Computer

Figure 1.13 Viewing the icon of the current VI.

In the above image, the icon is seen in the red box, and in the blue one, the shape the connector terminal has, which shows us how we can connect the elements in the Front Panel. Using left double click, the icon editor is opened, and it can be personalized.

1.5 Creating a VI



LabVIEW
File Operate Tools Help
LabVIEW™ 2024 Q1
Create Project
Recent Project Templates
Blank Project
Open Existing
All Recent Files
EnglishLabs.lvproj
test.lvproj
Lab8Lcd.lvproj
GenSemnal.lvproj
Untitled Project 1.lvproj
C:\Users\IonelBaciu\Desktop\Sad\T
C:\Users\IonelBaciu\Desktop\Sad\T
Fig1_2.vi
1/8
Get Started With LabVIEW
Access the LabVIEW tutorial and learn the basics of graphical programming. Get an overview of the environment, debugging, where to find examples, and more.
Read more
Find Drivers and Add-ons
Connect to devices and expand the functionality of LabVIEW.
Community and Support
Participate in the discussion forums or request technical support.

Figure 1.14 Creating a project.

The first step in developing a LabVIEW application is to create a project that will be saved under a particular name, the next being the creation of a new VI.

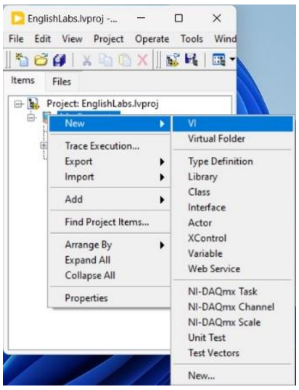


Figure 1.15 Creating a new VI.

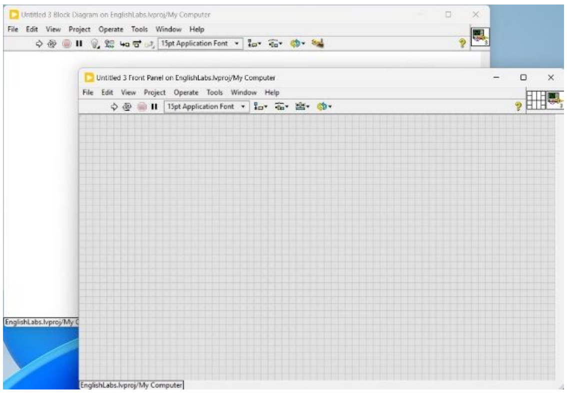

Untitled 3 Block Diagram on EnglishLabs.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Untitled 3 Front Panel on EnglishLabs.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
EnglishLabs.lvproj/My Computer

Figure 1.16 Front Panel and Block Diagram for a new VI.

1.6 Applications

1.6.1 Example 1 – implementing an application using Boolean elements and functions

$$result = \overline{(a \cdot b) \oplus (c + d)} \qquad 1.1$$

In the Block Diagram, from functions palette, select Boolean functions. From there, with drag & drop, select **AND, OR, XOR** and **NOT** blocks.

Using the wiring tool from tools palette, connect the blocks as seen in Figure 1.17.

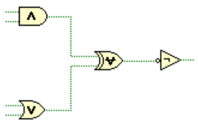
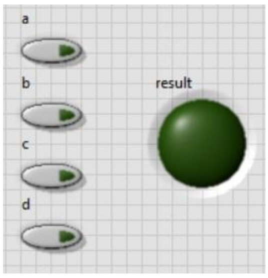


Figure 1.17. Wiring the first example.



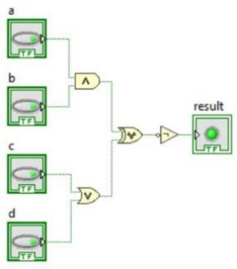


Figure 1.18 Front Panel and Block Diagram of the first example.

In the Front Panel, from the control palette, select Boolean elements subpalette. Then, add, using drag & drop, four Push Buttons and Round LED indicator. The values for the push buttons will be set by pressing each button, for the variables **a, b, c, d** (a lit push button has a **true** value).

The application can be run by pressing the Run button, to see the result.

1.6.2 Example 2 – implementing an application using elements and numerical functions

$$result = (x + y) \cdot (z/w) \qquad 1.2$$

In the Block Diagram, select Numeric Functions Palette. From the functions, using drag & drop, we select the following blocks: **Add, Divide** and **Multiply**. Using the Wiring Tool from Tools palette, connect the three blocks, as seen in Figure 1.19. In the Front Panel, select numerical controls sub palette, which will place four Numeric Control elements for setting numerical values for the 4 variables (x, y, z, w) and a numeric indicator for the output. In the Block Diagram, connect them appropriately.

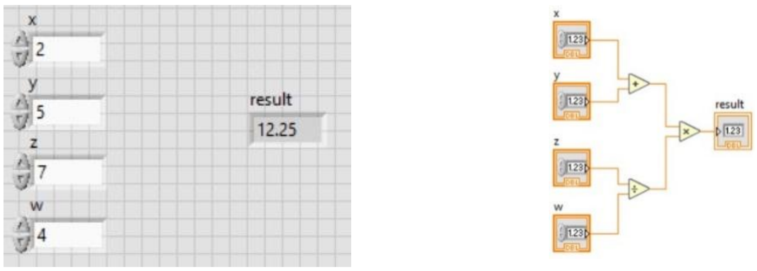


Figure 1.19 Front Panel and Block Diagram for the second example.

1.6.3 Example 3 – using and representing different data types

In this example, a warning generation application will be implemented, for exceeding the maximum or the minimum supply voltage value. This application needs: three **Numeric Control** elements, for selecting the minimum, maximum and current value thresholds, a String **Indicator**, for a text message and a Round **LED** for indicating the error state.

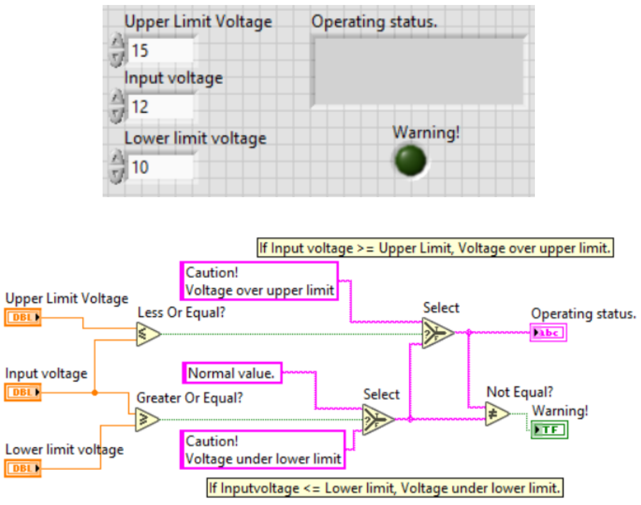


Figure 1.20 Front Panel and Block Diagram for the third example.

In the Block Diagram, add from the Comparison functions palette the following elements: **Less Or Equal?**, **Greater Or Equal?**, **Not Equal** and two **Select**.

Text messages that will be displayed by the Functioning state Indicator, are given by three **string constants** (Functions Palette, String Subpalette).

1.7 Questions and Exercises

1. Fill in the truth table for equation 1.1 and check, according to this table, the results obtained by the first example.

a	b	c	d	$x = a \cdot b$	$y = c + d$	$z = x \oplus y$	$\bar{z}$
0	0	0	0				
0	0	0	1				
0	0	1	0				
0	0	1	1				
0	1	0	0				
0	1	0	1				
0	1	1	0				
0	1	1	1				
1	0	0	0				
1	0	0	1				
1	0	1	0				
1	0	1	1				
1	1	0	0				
1	1	0	1				
1	1	1	0				
1	1	1	1				

2. In the first example, as well as in the second one, determine which operation will be executed first.
3. Implement the function *result = [(x/y) * (z-w)] + q.*
4. What can be noticed when representing different data types?

2 Programmable Structures in LabVIEW: For and While

2.1 Definitions

In LabVIEW, the programmable structures contain sections of graphical code and control how and when the code inside them is executed. The most common programmable structures are While, For and Case. They are used for running code sections several times or running them based on certain conditions. In LabVIEW, the structures can be found in *Functions – Structures* palette*.*

2.2 While structure

Like DO or REPEAT – UNTIL loops from the usual programming languages, **While** structure repeats the code (subdiagram) inside it until a specific condition is fulfilled. This subdiagram is executed at least once.

Figure 2.1 shows the components of a **While** structure. The **While** structure executes the code subdiagram until the condition terminal (bottom-right) which is an input terminal, receives a certain boolean value. The states of the condition terminal can be **Stop if True** or **Continue if True**. When the condition terminal is in **Continue if True** state, the While loop is executed until it receives a False condition. The iterations terminal (bottom-left) contains the number of iterations that have passed. The iterations counter starts from 0.

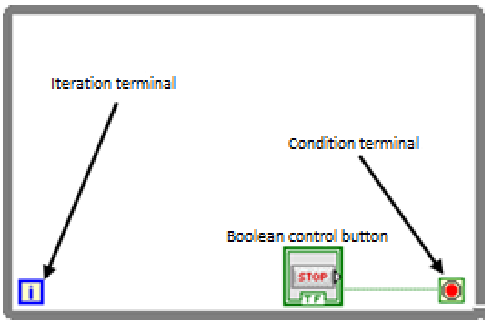


Figure 2.21 While structure components.

Changing the state of the condition terminal is done by left clicking on its surface.

Transmitting data from one iteration to another is done with shift registers. These shift registers are similar to static variables from C/C++. They are added with a right-click on the edge of the While loop and selecting ***Add Shift Register***.

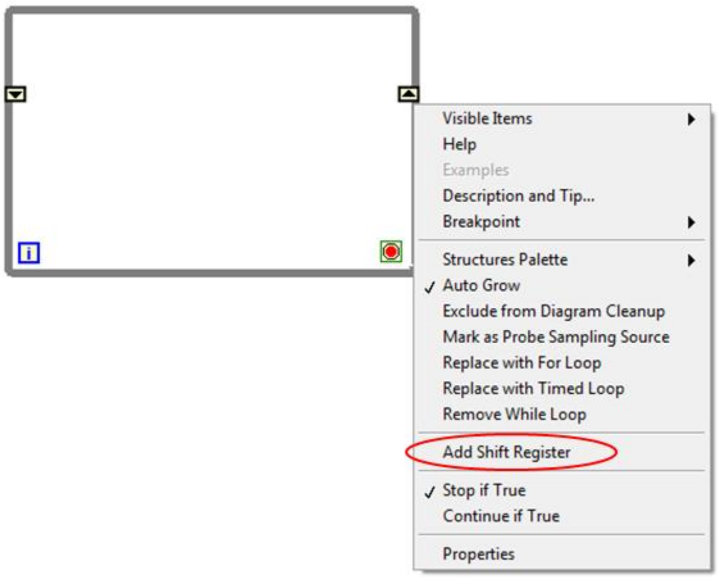


Figure 2.22 Adding a shift register.

Because the VI checks the condition terminal at the end of each iteration, the While loop gets executed at least once. The VI is not functional if the condition terminal is not connected.

2.3 For structure

A **For** structure executes the code inside it, for a known number of iterations. For structure has as main components:

- ***The count terminal*** – input terminal – specifies the number of executions for the code inside the structure. Routing a constant with negative or zero value prevents code execution.

- ***The iteration terminal*** – output terminal – indicates the number of executed iterations. For the first iteration, its value is 0.

Initial elements of a **For** structure can be observed in Figure 2.3.

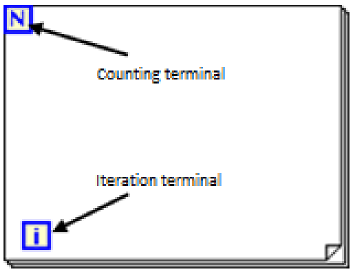


Figure 2.23 Main elements of a **For** structure.

For passing data from one iteration to the next, one must add shift registers, like the one described in Section 2.2.

Apart from its functionality of repeating code, a For structure can also be used for creating arrays, with its indexing properties. In Figure 2.4, the difference between the tunnels that leave the For loop is given by the indexing mode.

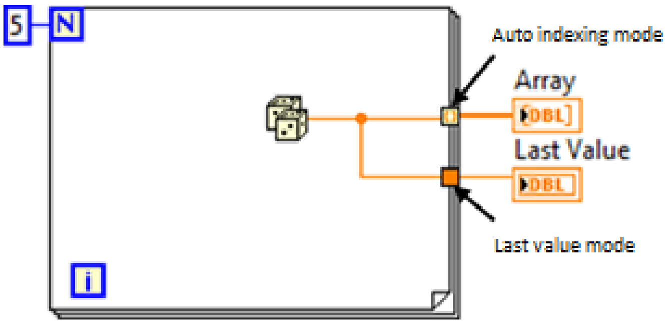


Figure 2.24 Indexing properties.

Selecting the output mode of the data is done with a right click on the tunnel terminal (shown with the two arrows in Figure 2.4) and selecting ***Tunnel mode*** field**.**

2.4 Applications

2.4.1 Example 1 – While structure

The first example is creating an array with the help of the iteration terminal of a While structure, creating a sum result for the elements in each iteration, and displaying the result of the last iteration. The Block Diagram and the Front Panel are presented in the next figures.

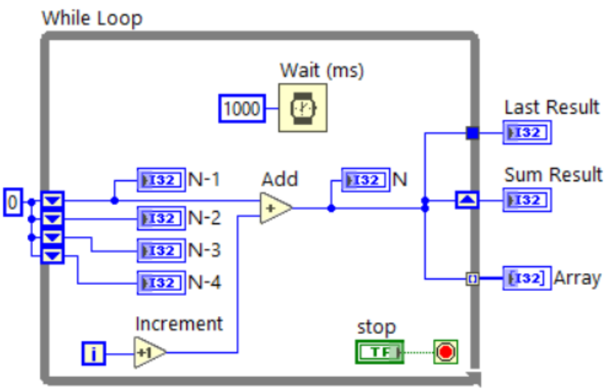


Figure 2.25 Block Diagram of the first example.

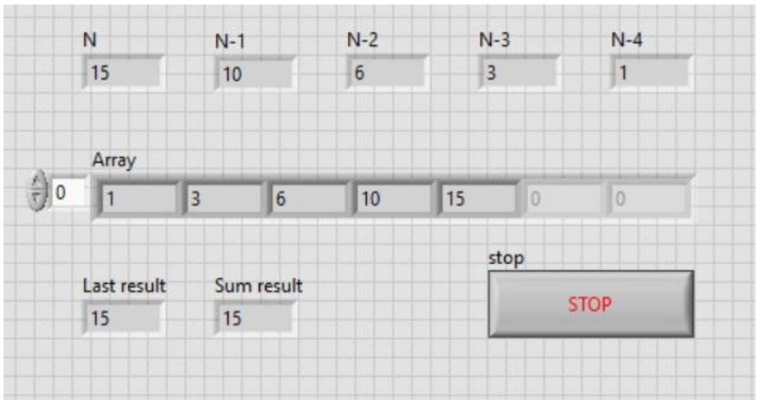


Figure 2.26 Front Panel of the first example.

Position the mouse above the first created Shift register until the Resize option is activated and expand it for creating multiple entities of the Shift Register at the input of the While loop.

Question: Notice the fields “Sum Result” and “Last Result”. When are they populated?

2.4.2 Example 2 – For structure

The second example repeats the functionality of the first example but using a For loop instead of the While Loop.

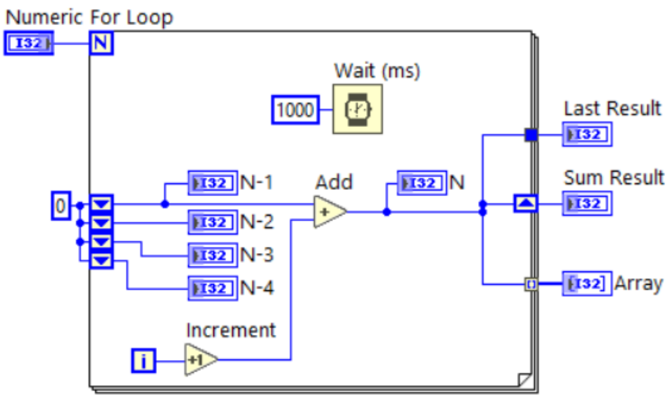


Figure 2.27 Block Diagram of the second example.

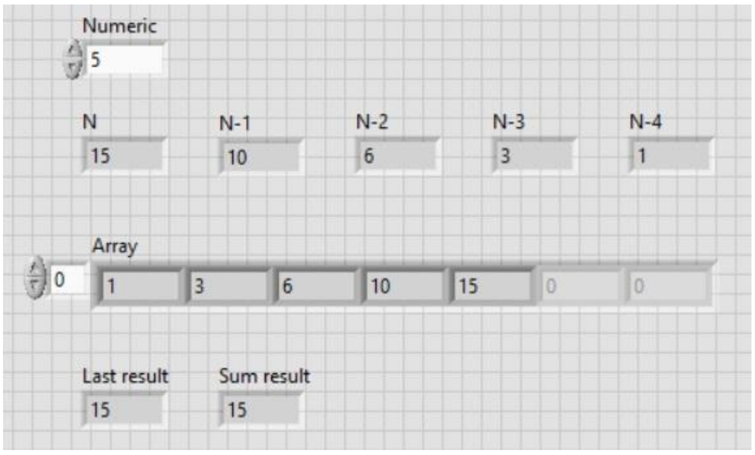


Figure 2.28 Front Panel of the second example.

Question: What are the differences between the first and second example, specific to this application?

2.4.3 Example 3 – transmitting data through local variables

In this example the execution of two while loops is shown, in parallel, using local variables. Their purpose is to read/write data in the controls or indicators placed in the front panel. The final Block Diagram is shown in Figure 2.9.

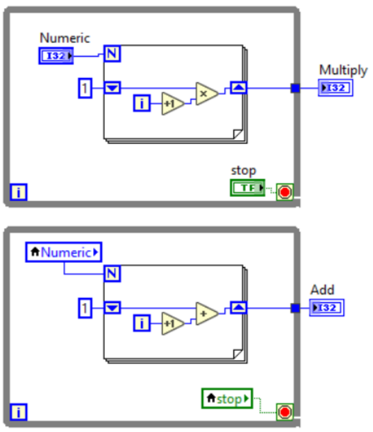
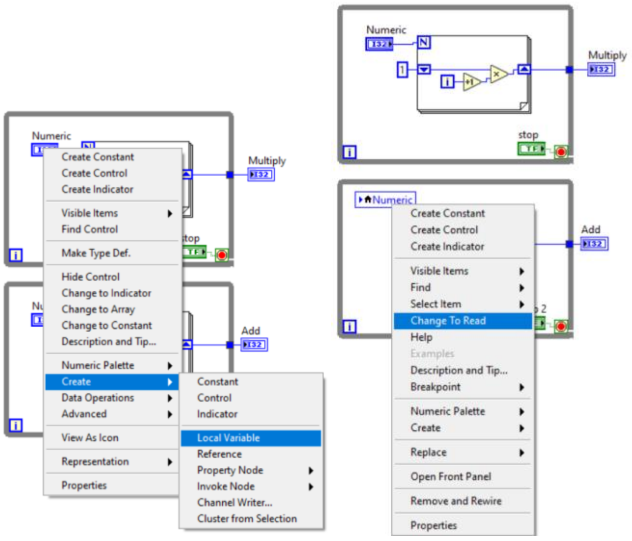


Figure 2.29 Block diagram for the third example.

It can be observed that for the Numeric and Stop controls, local variables were created for transmitting the value from the first structure to the second one. The steps for creating the local variables are shown in Figure 2.10. a). and b).



a). b).

Figure 2.30 Creating local variables for “Numeric” and “stop”.

When creating a local variable, its implicit state is in writing mode, thus, for reading the data, we must select “Change to Read” (Figure 2.10. b).

2.5 Questions and exercises

1. Which is the minimum number of iterations for a While structure?
2. Which is the value of the For Structure iteration terminal, if N = 5?
3. What is the execution order of the loops in example 3?
4. Design an application for generating the factorial value of a natural number N.

3 Programming Structures in LabVIEW: Case and Flat Sequence

3.1 Case structure

In LabVIEW, a **Case** structure contains two or more subdiagrams (cases), of which only one is executed at a time, depending on the value the structure has at its input terminal.

This structure is the equivalent of the „if/then/else” statement or a „switch/case” instruction in C++ programming.

The tag of the Case selector structure is located on the top of the block and contains the name of the currently selected case. On both sides of the tag, increment/decrement arrows can be found. With the help of these tags, the defined cases can be viewed. In the next figure a generic Case structure is presented.

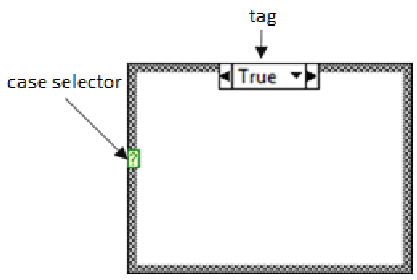


Figure 3.31 The components of a **Case** structure.

The Case structure can be found in **Functions Palette -> Structures -> Case Structure**. When placing a **Case** structure on the Block Diagram of a VI, the case selector will have the default data type of Boolean and the number of available cases will be two, corresponding to True and False. To create a control for this terminal, right click it and then select **„Create Control”**. At this moment, on the Front Panel of the VI a button will appear, which will have two states, True or False, corresponding to the tags of the **Case** structure. The mechanical action of the Boolean control can be modified with the help of a right click, as seen in the Figure below.

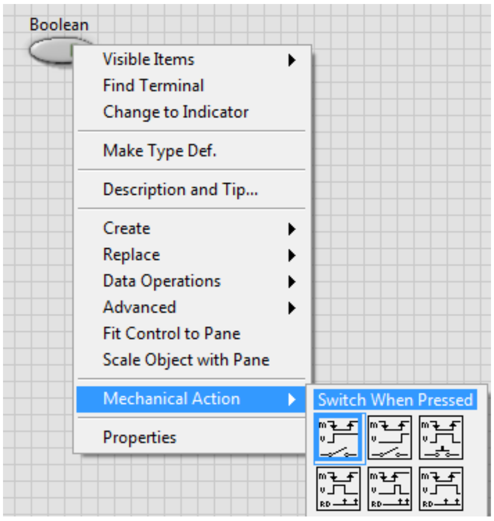


Figure 3.32 Modifying the mechanical action of a Boolean control.

The available mechanical actions are as follows:

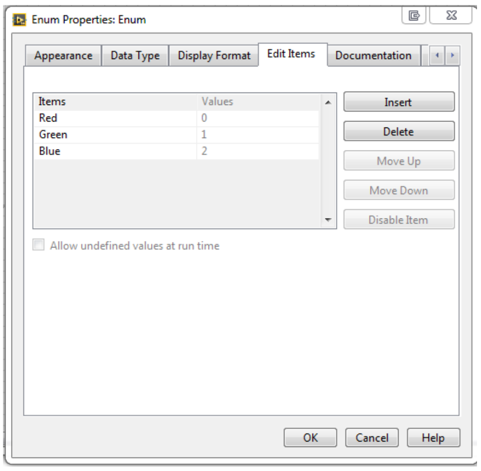
- “Switch When Pressed”,
- “Switch When Released”,
- “Switch Until Released”,
- “Latch When Pressed”,
- “Latch When Released”,
- “Latch Until Released”.

The difference between these states is intuitive, as seen in Figure 3.2. The state “Switch when Pressed” will change the state of the button to False from True when the button is pressed, “Switch when Released” will cause a state change then the button is released.

The Case selector determines which of its subdiagrams will be executed, depending on its entry value. The date type for the entry variable can be Boolean, string, int, enum or an error cluster. Additional cases can be added, depending on the data type, by right clicking the structure and selecting „Add Case...”.

To define a case selector with type Enum, place an Enum by right clicking on the Front Panel and then selecting **Ring & Enum -> Enum before**

routing anything to the selector terminal. Defining of the Enum states can be done on the Front Panel by right clicking the control and selecting „Edit Items”. The properties window shown in Figure 3.3 will appear.

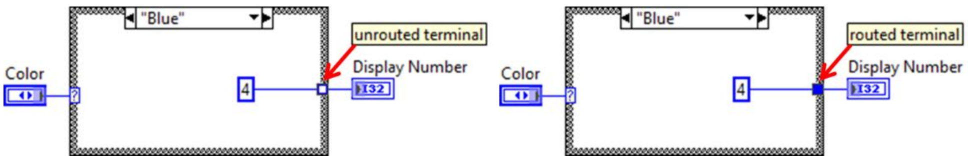


Items	Values
Red	0
Green	1
Blue	2

Figure 3.33 Modifying the properties of an Enum type controller.

After defining the states that the Enum control can take, it can be routed to the Case selector. At this moment the Case selector tags will automatically **take** the states defined in the enum but will **not** automatically **create** all the states. Using the option „Add Case...” (accessible by right clicking on the edge of the case structure) all the necessary cases can be created to equal the number of cases defined in the Enum.

After adding the necessary graphical code to the subdiagram of the Case structure, it is necessary to connect all the output terminals in **all subdiagrams.** If one of the terminals remains unrouted, this will look like the one in Figure 3.4. a). and it will cause an error when running the VI. Figure 3.4. b). shows a terminal that is routed for all cases.



a). b).

Figure 3.34 States of an output terminal for a Case structure.

3.2 Flat sequence

A Flat Sequence structure contains one of more subdiagrams or frames which are executed in a sequential order. This is used to ensure the execution of one piece of code before the other. The data flow in case of a flat sequence differs from the data flow of other structures. The frames of a flat sequence are executed in order, from left to right when all the data routed to the frame is available. The input data of one frame is dependent on the output data of the previous frame.

The Flat Sequence can be placed on a Block Diagram of a VI by selecting **Functions Palette -> Structures -> Flat Sequence Structure**. By default, the Flat Sequence structure contains one frame, however, but additional frames can be added by right clicking on the edge of the frame and selecting „Add Frame...”. A generic Flat Sequence structure as well as the adding of frames can be seen in Figure 3.5.

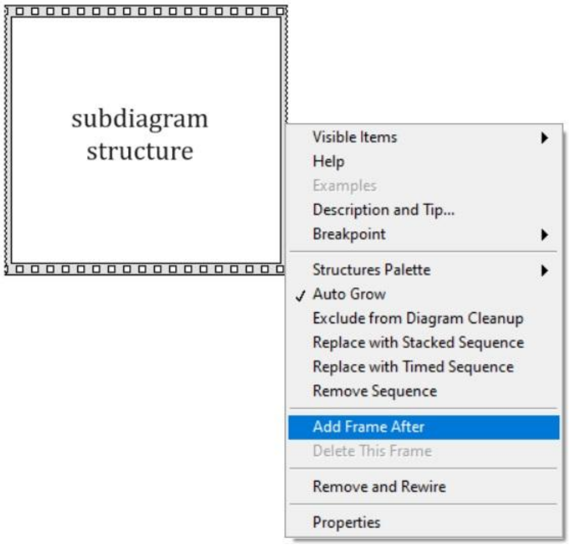


Figure 3.35 The Flat Sequence.

A Flat Sequence structure is executed starting with frame 0 (leftmost one), frame 1, frame 2 until the last frame (rightmost one) is executed. This structure does not finish its execution and does not return data until the last frame is executed.

3.3 Applications

3.3.1 Example 1 – Case structure

The first example shows the display of a constant specific to a certain case. The Case Structure contains 7 cases, corresponding to the colors of the rainbow (ROYGBIV) (0-6). For each case a different constant is displayed. The Block Diagram for this first example can be seen in Figure 3.6.

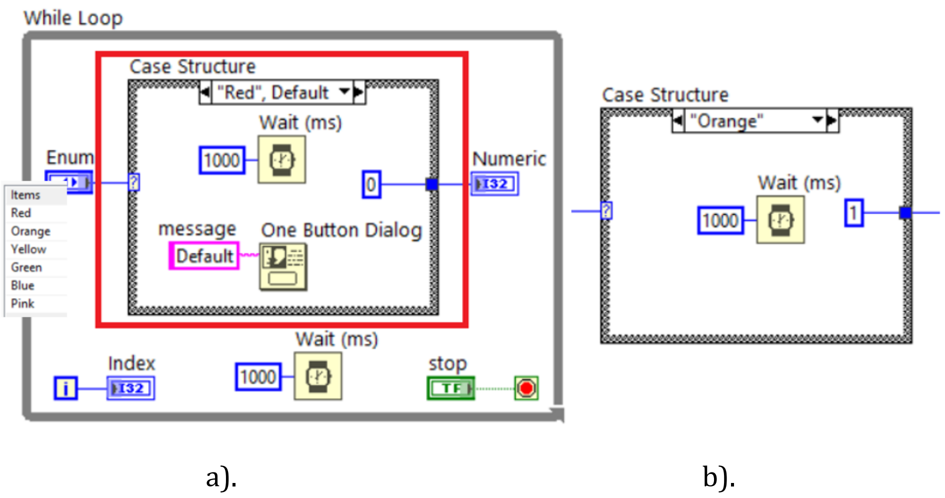


Figure 3.36 a) Block Diagram for Example 1 and b) the content of next cases of the Case Structure.

The Block Diagram shown in Figure 3.6 a). represents a Case structure inside a While loop. This implementation was chosen to ensure the continuous running of the application when pressing the „Run” button and is the equivalent of a **state machine**.

3.3.2 Example 2 – Flat sequence

The second example shows the operation of a Flat Sequence structure. The execution order of the frames will be from left to right and the value of the LED 1 variable will be modified with the help of a local variable.

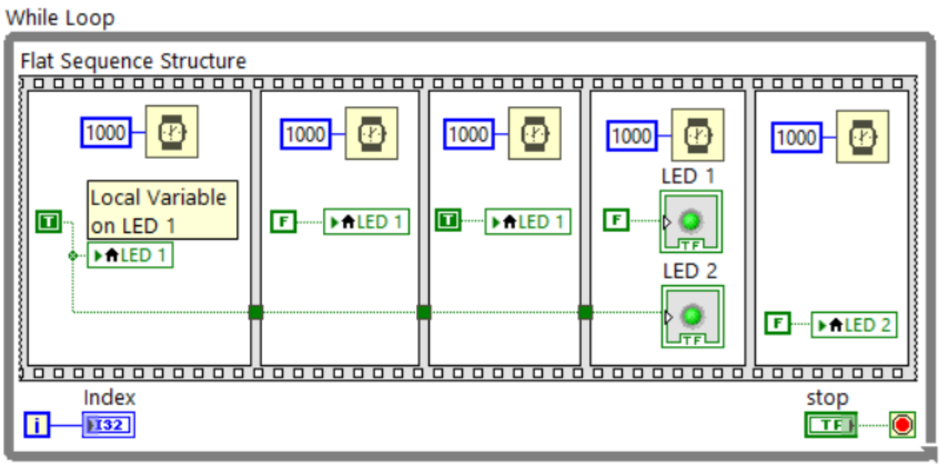


Figure 3.37 Block Diagram for the second example.

3.4 Questions and exercises

1. What is the information displayed on the Front Panel of the first example? What about the second example?
2. What is the effect of the Wait block inside the While structure in the case of the first example?
3. When is the LED 2 variable updated in the case of the second example?

4 Arrays, Matrices and Clusters

4.1 Definitions

Arrays are collections of data of the same type. They can have one or more dimensions forming arrays or matrices. In LabVIEW, arrays can have all data types. The only limitation is that arrays of arrays cannot exist.

Each element of an array can be accessed through its index. The index is between 0 and N-1, where N_n is the total number of elements. A 1D array is shown in Figure 4.1.

index	0	1	2	3	4	5	6	7	8	9
vector 10 elements	1.2	1.7	1.1	1	1.2	3.1	2.5	2.4	1.2	1.6

Figure 4.1 Array of 10 elements.

Arrays in LabVIEW are 0-based indexed.

Unlike arrays, **clusters** are elements that can contain different data types. They are similar with structures from the usual programming languages.

4.2 Creating input arrays

Input arrays are of two types - those that can be accessed from the Front Panel and constant vectors, which only have a correspondent in the Block Diagram.

4.2.1 Control and indicator arrays

4.2.1.1 Manual mode

From the controls palette of the Front Panel **Array & Cluster** group can be selected.

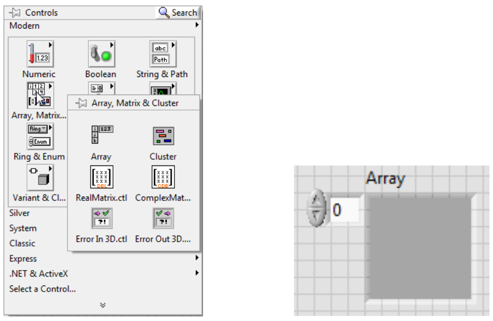
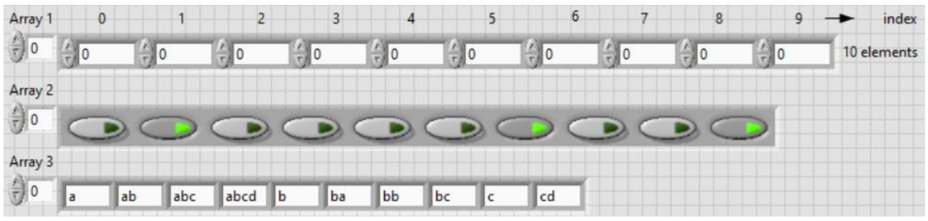
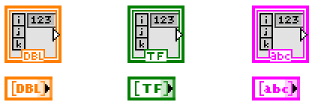


Figure 4.2 Controls palette for vectors, matrices, and structures available in the Front Panel.

From this palette, with drag & drop, we can select **Array** block. Inside the created field, any type of element can be added, for example numeric controls, boolean controls or string controls. Also, an indicator array can be built in the same manner, by replacing the controls with indicators. The first dragged element decides the data type for the whole array.



a).



b).

Figure 4.3 a). Array 1 is numeric, Array 2 is Boolean, and Array 3 is a string. b). Correspondent for each array on the Block Diagram.

The default appearance mode for an array in the Block Diagram can be changed, by right clicking on the icon and selecting **View as Icon**.

In the Front Panel, the element on the left (highlighted in Figure 4.4) indicates the index of the first **visible** element of the array. This is useful for viewing a certain element from a certain index, when the array is large.

Modifying this first element **does not** lead to modifying the array size (number of elements).

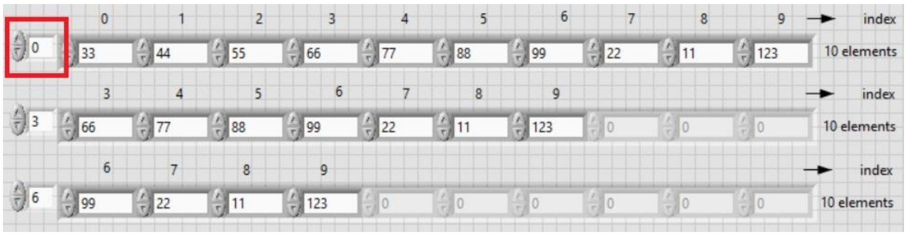


Figure 4.4 Red box: array index.

Sometimes it is necessary to use matrices. A matrix is a multidimensional array, which requires the existence of two indexes: an index corresponding to the number of lines and another index corresponding to the number of columns. Both indexes are zero-based.

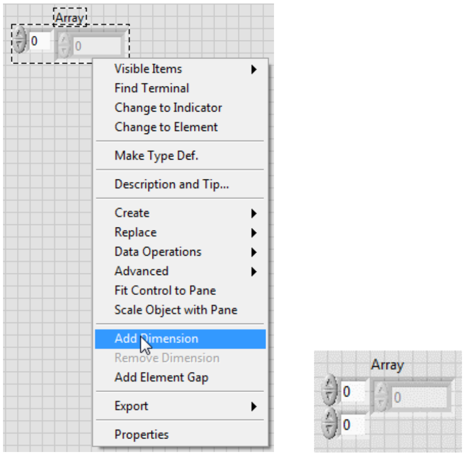


Figure 4.5 Adding a new dimension to obtain a matrix.

4.2.1.2 *Automatic mode*

The loops studied in Chapter 2 can index and accumulate arrays automatically, this feature being called self-indexing. However, there is a difference between the two, the **While** loop and the **For** loop, which consists in the number of elements allocated by the processor to create that array. In the case of the **While** loop, the processor will allocate a **maximum** number of elements, the limit being the available memory. In the case of the **For** loop, the number of elements is given directly by **N**, thus for the automatic generation of arrays the **For** loop is used, the **While** loop being used only in situations when memory allocation is not an issue.

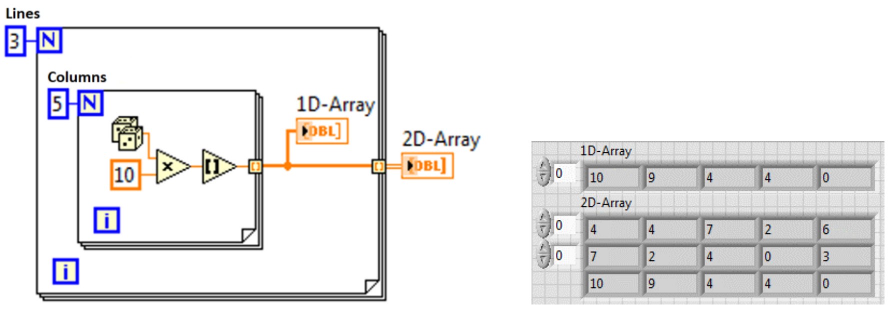


Figure 4.6 Automatic generation of a 2D array using self-indexing.

The inverse operation is valid as well, in which at the input of a **For** loop we connect an array with a certain number of elements, X. In this situation it is necessary to know that **if N is greater or equal to X, the number of iterations of the For loop will be set to the number X. If N is less than X, then the number of iterations will be N.**

4.2.2 Constant arrays

These arrays can be defined and modified only in the Block Diagram. Creating arrays of constant elements is similar to creating the array of control elements, only they will be generated in the Block Diagram.

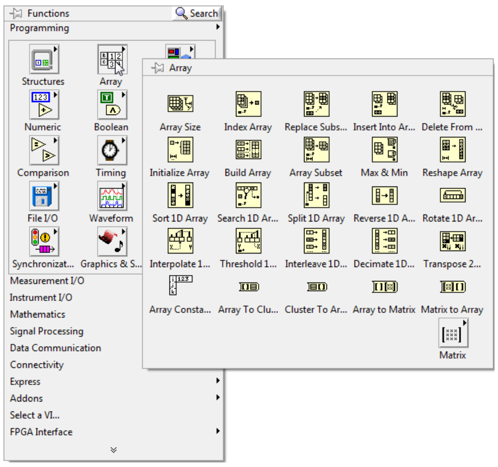


Figure 4.7 Array functions subpalette.

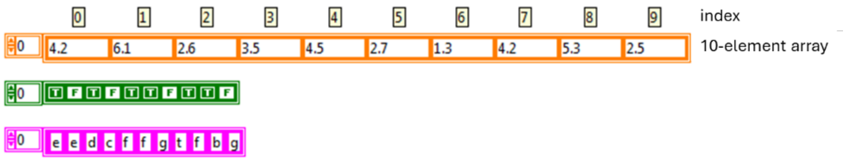
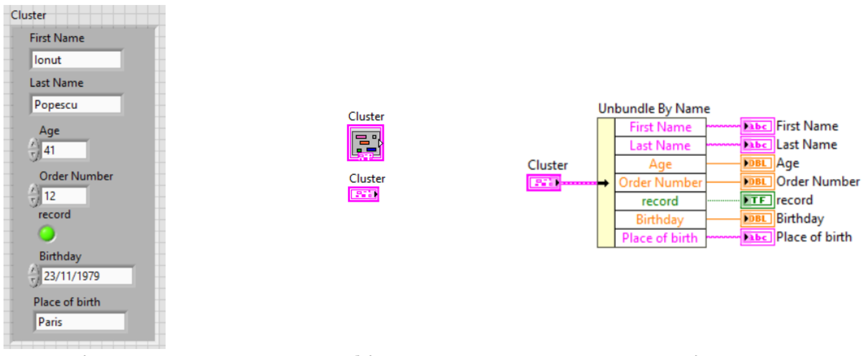


Figure 4.8 Types of arrays of constant elements.

4.3 Creating clusters

Clusters are used to organize data. Their use facilitates reading diagrams of complex applications by reducing the number of existing connectors.

The method of creating clusters is like the creation of arrays. From the controls palette in Figure 4.2, using drag & drop the **Cluster** element can be selected. Within the **Cluster** object, using drag & drop, all the necessary elements can be inserted, regardless of the datatype. They can be numerical controls, boolean, string control, indicators, etc., all placed in the same Cluster.



a). b). c).

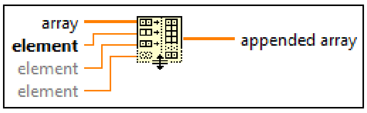
Figure 4.9 Cluster a). Front Panel; b). Block Diagram equivalent; c). accessing each cluster element.

Both in the arrays and clusters cases, we cannot have, inside the same object controls, indicators **and** constants. Different objects must be created for each type.

4.4 Applications

4.4.1 Example 1 – build array

The example presented below uses the Build Array function to create arrays from various input types. Build Array function has two options: a default one, which concatenates the inputs (adding an element at the end of another, resulting in a 1D array), and one for creating a 2D array, obtained by right clicking on the Build Array function, and de-selecting the option **Concatenate Inputs**. When placing the function on the Block Diagram, it has only one input available. Inputs are to the node by right clicking an input and selecting **Add Input** from the menu or by resizing the node.



When creating control and indicator elements in the Block Diagram for the Build Array function terminals, by default, they have only one element in the Front Panel. The array can be increased to the number of elements needed by resizing it, using the resize handle found on the edge of the array shell border.

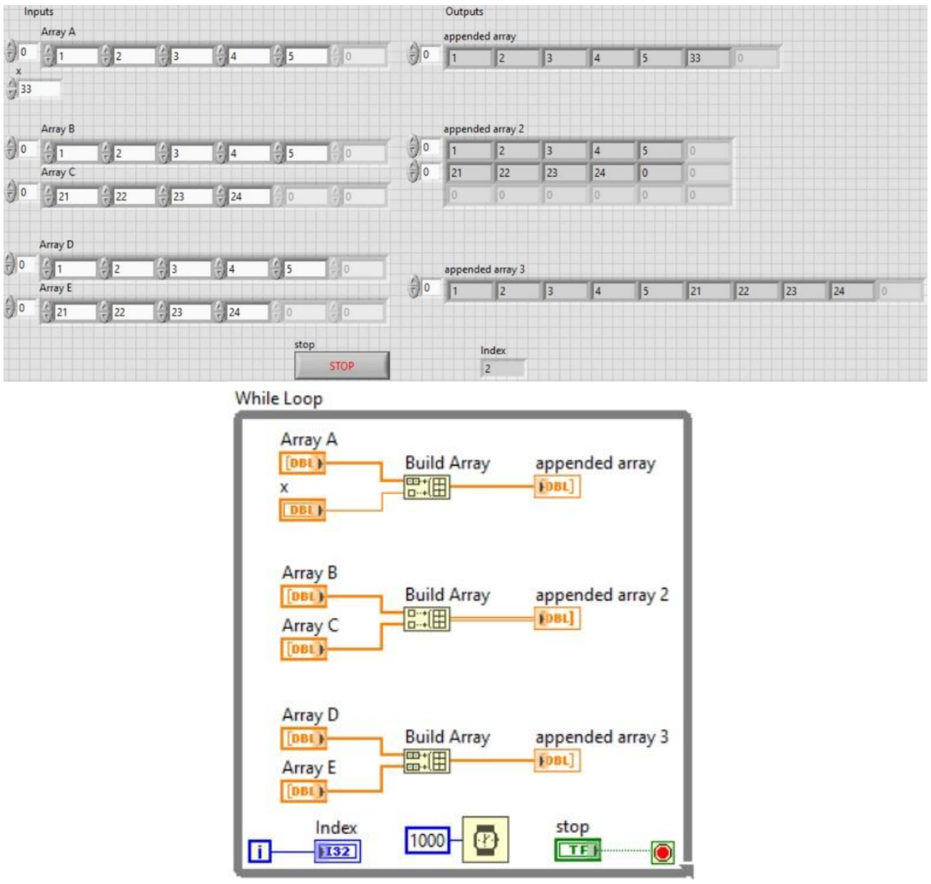


Figure 4.10 Front Panel and Block Diagram of the first example.

4.4.2 Example 2 – array operations

In this example, two array creation methods are implemented, one using the Index Array function and one using the **For** structure. By running this example, you can see the difference between the two creation methods as well as what happens when performing simple mathematical operations (add, subtract, multiply, divide) with them.

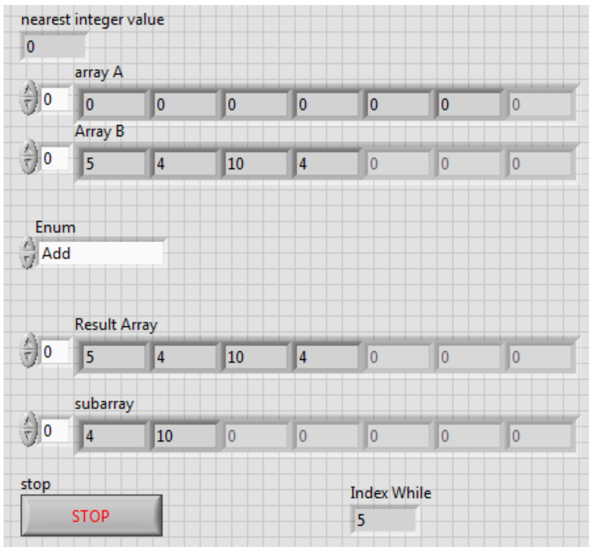


Figure 4.11. Front Panel for the second example.

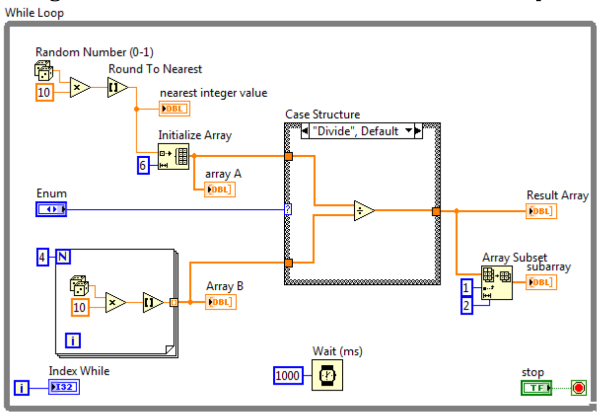
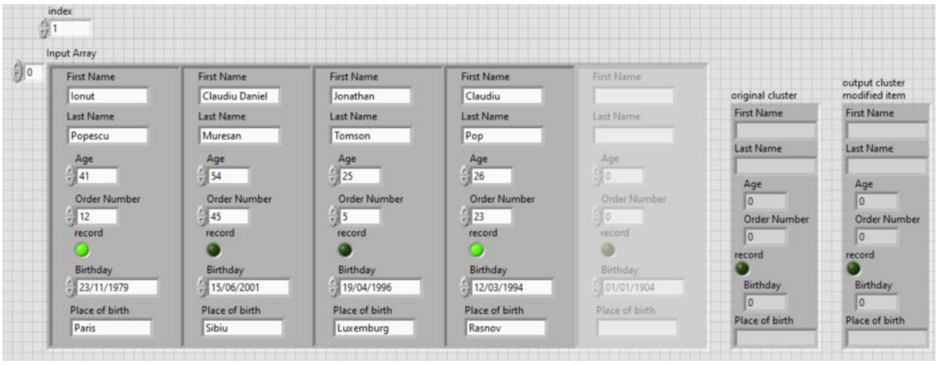


Figure 4.12. Block Diagram for the second example.

4.4.3 Example 3 – cluster operations

In this example, an operation for modifying the value of an element embedded in a cluster is presented. In Figure 4.13 an array of clusters can be seen, in which an array cell is identical with the cluster presented in Figure 4.9. The method for creating the input array is presented in paragraph 4.2.1.1, the first step is creating the cluster and after that, the cluster is dragged into an array field.


index
1
Input Array
0

First Name
Ionut
Last Name
Popescu
Age
41
Order Number
12
record
Birthday
23/11/1979
Place of birth
Paris

First Name
Claudiu Daniel
Last Name
Muresan
Age
54
Order Number
45
record
Birthday
15/06/2001
Place of birth
Sibiu

First Name
Jonathan
Last Name
Tomson
Age
25
Order Number
5
record
Birthday
19/04/1996
Place of birth
Luxemburg

First Name
Claudiu
Last Name
Pop
Age
26
Order Number
23
record
Birthday
12/03/1994
Place of birth
Rasnov

First Name
Last Name
Age
0
Order Number
0
record
Birthday
01/01/1904
Place of birth

original cluster
First Name
Last Name
Age
0
Order Number
0
record
Birthday
0
Place of birth

output cluster modified item
First Name
Last Name
Age
0
Order Number
0
record
Birthday
0
Place of birth

Figure 4.13 Front Panel for the third example.

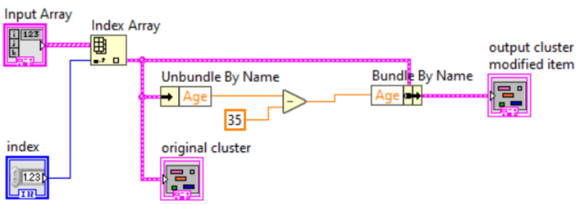


Figure 4.14 Block Diagram for the third example.

4.5 Questions and exercises

1. Develop an application in which, by using the Index Array block, extract:
   a. 1 element chosen by the user from a 1D array,
   b. 1 row chosen by the user from a 2D array,
   c. 1 column chosen by the user from a 2D array.
2. What is the purpose of the Array Subset block in example 2?
3. What is the difference between array A and array B in example 2?

5 Strings and I/O Files

5.1 Strings

A printable sequence of characters represents a **string**. They can be used for more than just writing text. For example, in the case of control instruments, numerical data is represented through string characters, which afterwards are converted to numerical values. In many situations, saving numerical data implies using strings, which means that those numerical values need to be converted to strings before writing them to a file.

Like for the other data types, for example Numerical or Boolean, LabVIEW provides a subpalette in the Front Panel for strings, with control and indicator elements. The subpalette is called String & Path.

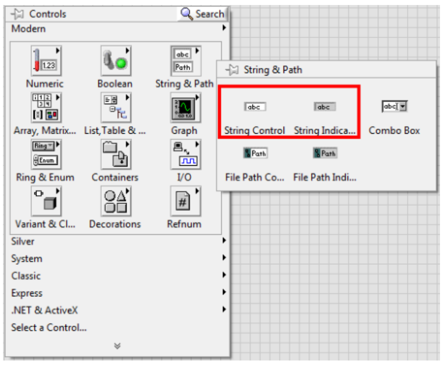


Figure 5.1 String control and indicator subpalette.

Unlike the Numerical elements and Boolean, for String elements we have the possibility to configure the display of data. The types available are Password Display, '\' Codes Display, Hex Display and Normal Display.

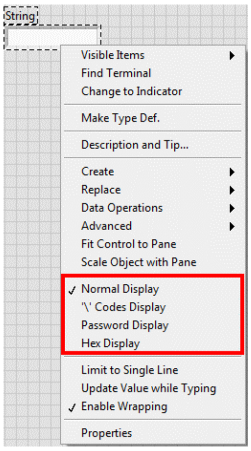


Figure 5.2 Choosing data display type.

In the Block Diagram, LabVIEW provides us with a functions subpalette just for strings.

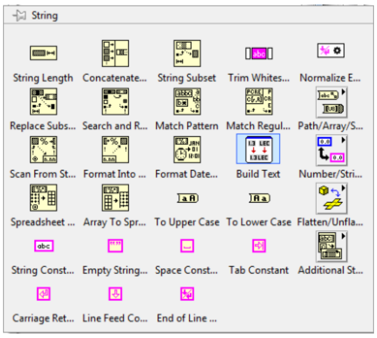


Figure 5.3 String functions subpalette.

There we find certain functions that perform a series of operations on strings and certain common constants. In addition, a subpalette for conversion between Number / String is available.

5.2 File I/O

Input and output files (I/ O) are used to read and save data on the storage media. LabVIEW provides us with a range of functions for these operations.

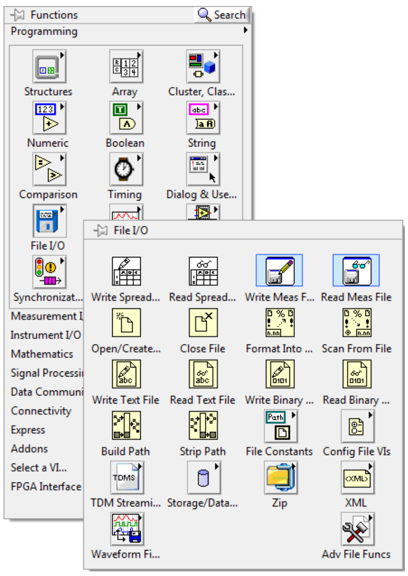


Figure 5.4 Functions subpalette for file operations.

Files are one of the types of resources available and can be accessed using the functions above. Resources are the addressable files, hardware, or network connection of that system. LabVIEW includes several features that allow access to these resources. The resources are recognized by the system's access routes to them (path) name, port, or another identifier.

One such application, in which we use resources, has a structure similar to the one shown below.

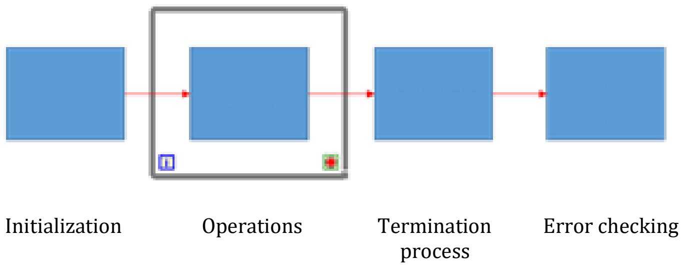


Figure 5.5 Basic structure for a resource application.

The initialization stage comprises specifying a path to a used resource (path) or name of the used device. Following this stage, LabVIEW creates a reference number (refnum), which is a unique identifier for each resource. Basically, refnum is a temporary pointer to the resource.

The termination process block releases the used resources. LabVIEW allocates memory for each object that is assigned to a refnum. When the process is finished, the memory is freed.

Access and I/O file operations are performed using applications that have a similar basic structure to the one presented in Figure 5.5.

LabVIEW can use or create the following file formats:

Binary - efficient, compact, allows reading random elements, but not eye-readable. It is used to read and write data with an increased speed, for example in DAQ applications.

ASCII - text is readable to the naked eye, the data is represented as strings.

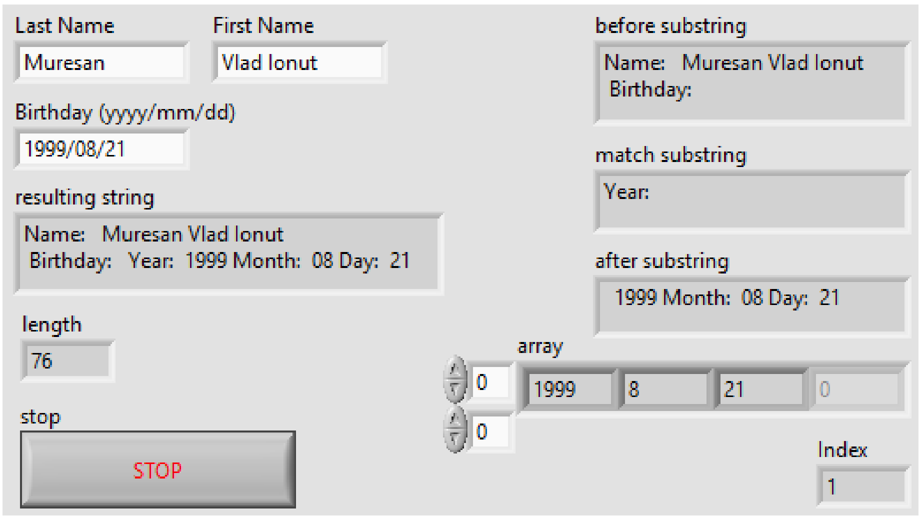
LVM - LabVIEW measurement data file is based on ASCII code; it is a text file delimited by Tab character. These files can be read by spreadsheet applications, LabVIEW default and applications like Excel.

TDMS - is a binary format specifically to NI and comprises both data and properties of such data.

5.3 Applications

5.3.1 Example 1 – generating strings

In this application, several methods of generating strings and string constants are implemented, using default TAB or End of Line, or created by the user.



Last Name: Muresan

First Name: Vlad Ionut

Birthday (yyyy/mm/dd): 1999/08/21

resulting string:
Name: Muresan Vlad Ionut
Birthday: Year: 1999 Month: 08 Day: 21

length: 76

stop: STOP

before substring:
Name: Muresan Vlad Ionut
Birthday:

match substring: Year:

after substring: 1999 Month: 08 Day: 21

array: 0 0 | 1999 | 8 | 21 | 0

Index: 1

Figure 5.6 Front Panel of the first example.

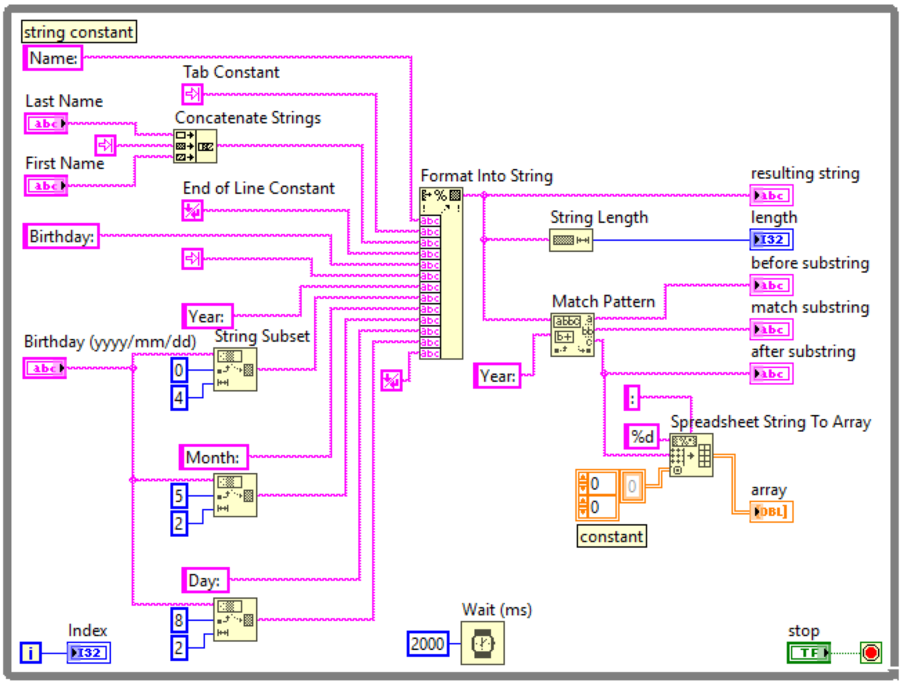
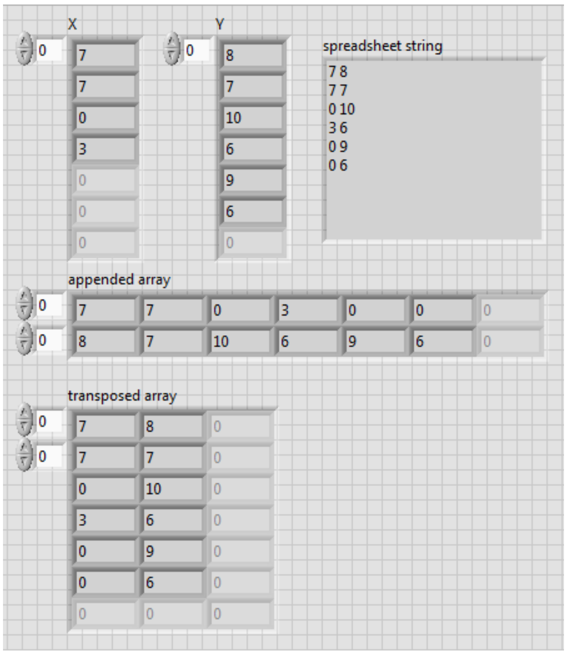


Figure 5.7 Block Diagram of the first example.

5.3.2 Example 2 – writing a string in an ASCII file

Several operations with arrays are demonstrated in this application, as well as writing all data to an ASCII file.



X

0
7
7
0
3
0
0
0

Y

0
8
7
10
6
9
6
0

spreadsheet string

```
7 8
7 7
0 10
3 6
0 9
0 6
```

appended array (index 0, 0)

7	7	0	3	0	0	0
8	7	10	6	9	6	0

transposed array (index 0, 0)

7	8	0
7	7	0
0	10	0
3	6	0
0	9	0
0	6	0
0	0	0

Figure 5.8 Front Panel of the second example.

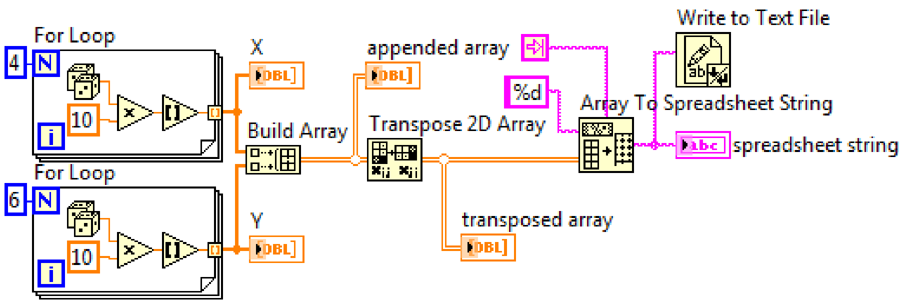


Figure 5.9 Block Diagram of the second example.

5.3.3 Example 3 – reading and writing data from/to a binary file

Binary file writing example. This application involves generating a vector of 10 elements which are then saved in a binary file.

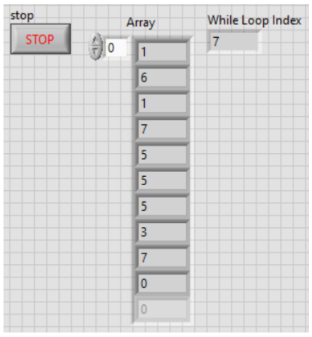


Figure 5.10 Front Panel of the third example, writing example.

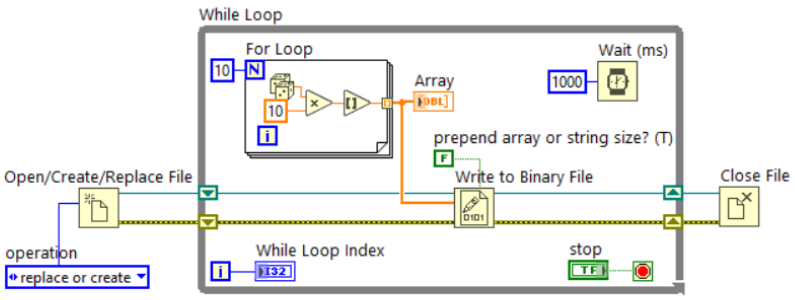


Figure 5.11 Block Diagram of the third example, writing example.

Binary file reading example. This application involves reading data from the previously created binary file.

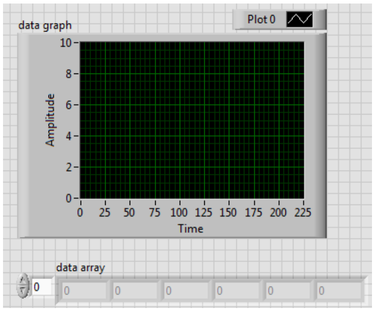


Figure 5.12 Front Panel of the third example, reading example.

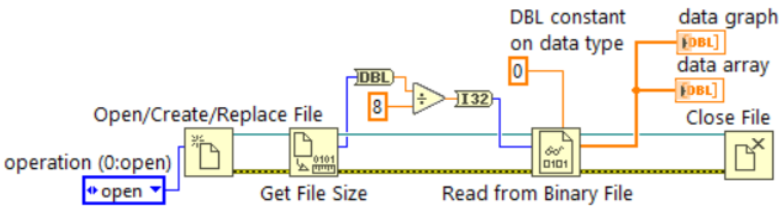


Figure 5.13. Block Diagram of the third example, reading example.

5.3.4 Example 4 – saving and viewing data in TDMS format (Technical Data Management Streaming)

This file format contains two types of data: data on the saved name and properties, and measurement data, saved in binary format.

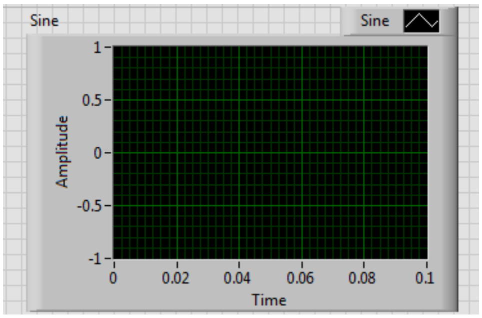


Figure 5.14 Front Panel of the fourth example.

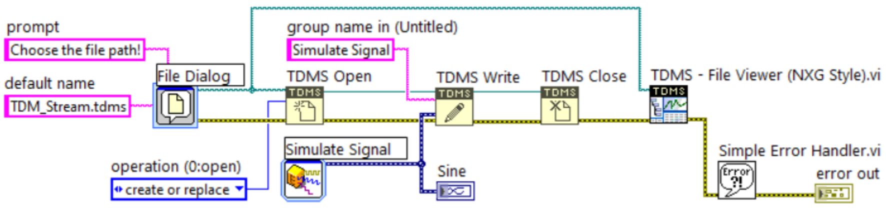


Figure 5.15 Block Diagram of the fourth example.

5.4 Questions and exercises

1. Improve the functionality of the second example, so it matches the structure in Figure 5.5.
2. What can be seen when looking at the data in the new file created in Example 3, situation 1?
3. How can the data saved in TDMS format be viewed?

6 Waveforms, Filters and Noise

LabVIEW provides us with a subpalette function in the Block Diagram called **Waveform**.

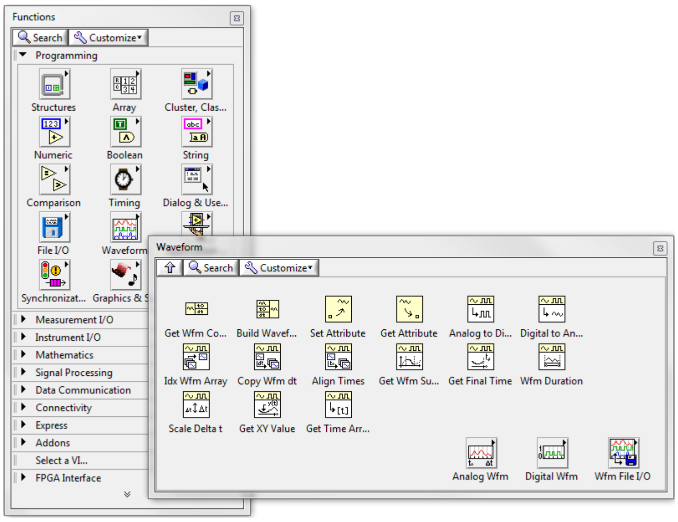


Figure 6.1 Functions subpalette for generating waveforms.

The items found in the subpalette are used to perform certain functions such as:

- generating analog, periodic (sine, rectangle, etc.), random (noise), and digital waveforms,
- extracting individual data elements of a waveform,
- editing individual data elements of a waveform,
- writing or reading a waveform in or from a file.

These waveforms can also be defined using models. Basically, we generate wave signals by their duration. We can control the signal's amplitude, phase, number of cycles, and number of samples. The palette used to define the generation functions, filtering, measuring, etc., is found in **Functions >> Signal Processing**.

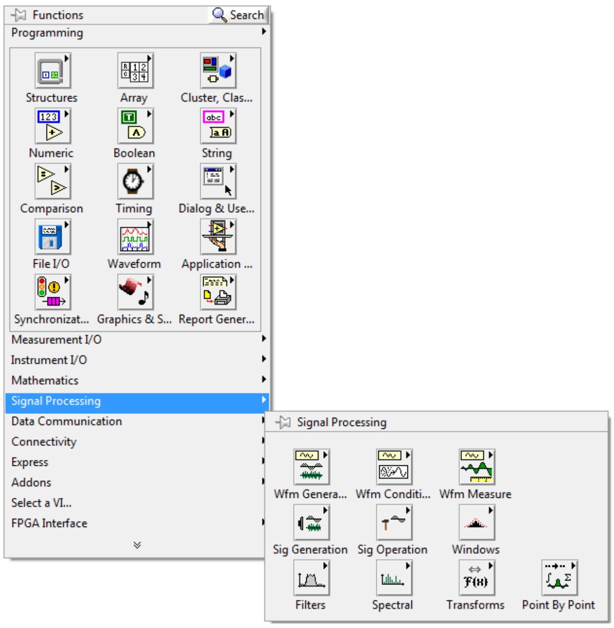


Figure 6.2 Signal Processing functions subpalette.

6.1 Measuring waveforms parameters

LabVIEW provides us with a subpalette function within a Block Diagram that helps in making measurements of signals in both time domain and frequency domain. You can perform:

- measurements of average values of signals (DC-Direct Current),
- measurements of effective values (Root Mean Square RMS),
- measurement of signals' amplitude and level,
- FFT spectrum (it returns phase and amplitude),
- FFT power spectrum, rise and fall times, the growth rate.

To access the measurement functions subpalette, select **Functions >> Signal Processing >> Measure WFM**.

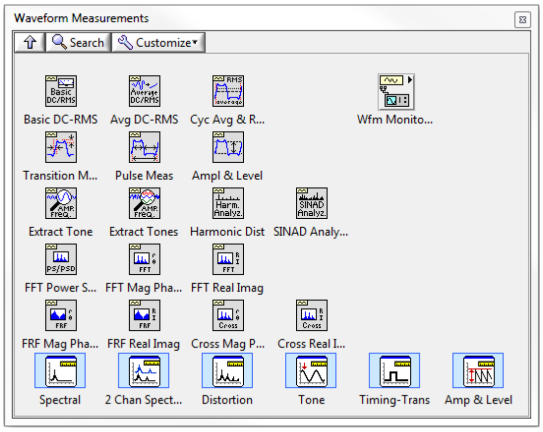


Figure 6.3 Waveform Measurements functions subpalette.

We can use the DC value to define the value of a static signal, or which varies slowly. DC measurements can have positive or negative values. The continuous DC level of a signal v(t) over the time interval from t_1 to t_2 is given by the equation:

$$V_{DC} = \frac{1}{t_2 - t_1} \cdot \int_{t_1}^{t_2} V(t) \cdot dt \qquad 6.1$$

, where t_2-t_1 is the integration time or measurement time. Therefore, DC value is the average of a signal value, calculated over a range of time.

The RMS level measurement is used when a representation of the energy is needed. Its value is always positive. RMS level of a continuous signal on a time interval (t_1, t_2) is given by the equation:

$$V_{RMS} = \sqrt{\frac{1}{t_2 - t_1} \cdot \int_{t_1}^{t_2} V^2(t) \cdot dt} \qquad 6.2$$

, where t_2-t_1 is the level of integration or measurement time. So, the RMS is the effective value of a signal measured at a time.

6.2 Noise signals

Uniform white noise - a noise signal, which is not repeated, and for which the spectral energy/Hz is independent of frequency. Its spectrum looks flat on the display of a spectrum analyzer.

Gaussian white noise - a noise signal with a Gaussian distribution of its instantaneous amplitude values. The frequency spectrum of such a signal is flat and has equal values at all frequencies.

Pseudorandom noise - the signal spectrum is flat. The noise is generated using a digital feedback shift register, so the noise sequence is repeated after a given number of samples. Also, since the signal is repeated, it is a discrete frequency spectrum, with a spectral component at the frequency N*F, where F = 1/T, T being the length in seconds of the sequence. Because the signal is repeated, it is called pseudorandom noise. True random noise does not repeat and has a continuous spectrum.

Noise signals can be used to perform frequency response measurements or to simulate certain processes.

a. The term **White -** ideal white noise has equal power per unit of bandwidth, resulting in a flat power spectrum. Thus, the power in the frequency range of 100Hz to 110Hz is the same as in the range of 1000Hz to 1010Hz. In practical measurements, achieving a flat power density spectrum would require an infinite number of samples. Thus, when we measure the power spectrum of the white noise, the values are usually mediated.
b. The terms **Uniform and Gaussian** refer to the probability density function of the amplitudes corresponding to the time domain noise samples. For the white noise, the probability density function is uniform within a specified interval. So, all values of the amplitudes within certain limits are equally probable.

Pseudorandom noise is a sum of sinusoidal signals with the same amplitude but with random phases. This noise does not have power at all frequencies, only at discrete frequencies corresponding to the harmonics of the fundamental frequencies. However, the noise level at each discrete frequency is the same.

6.3 Digital filters

Signal filtering is a basic operation in processing information transmission through noisy environments. Digital signal filtering means the signal spectrum processing represented by sequences of numbers at discrete time intervals by means of software implementations of algorithms. Signal filtering in the presence of noise can be done using the filters described below.

6.3.1 Butterworth filter

This filter has a monotonic attenuation characteristic of type maximum flat. The transfer characteristic of such a filter is shown below:

$$|F(j\omega)|^2 = \frac{1}{1+\left(\frac{\omega}{\omega_C}\right)^{2\cdot n}} \qquad 6.3$$

The graph of the transfer function for several values of n is shown below:

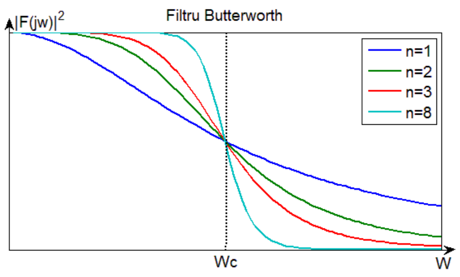


Figure 6.4. Butterworth filter transfer characteristic.

6.3.2 Chebyshev filter

This type of filter is specified by the equation:

$$|F(j\omega)|^2 = \frac{1}{1+\varepsilon^2\cdot {V_n}^2\cdot\left(\frac{\omega}{\omega_C}\right)} \qquad 6.4$$

where ε is the ripple and $V_n(x)$ is a Chebyshev polynomial of order n that can be generated by the recurrence formula:

$$\begin{gathered} V_n(x) = 2 \cdot x \cdot V_{n-1}(x) - V_{n-2}(x); \\ \dots \\ V_1(x) = x \\ V_0(x) = 1 \end{gathered} \qquad 6.5$$

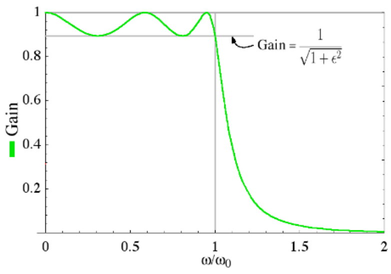


Figure 6.5. Chebyshev filter transfer characteristic.

6.3.3 Elliptic filter

Elliptic filter is based on the properties of Jakobi elliptic function. This function, denoted with $s_n(\omega)$, is a periodic double function of complex variable ω and analytical in the plan u except for simple poles. Since the function is double periodic, the base pair of the two zeros and poles is repeated infinitely along the axes x and y.

$$|F(j\omega)|^2 = \frac{1}{1 + \varepsilon^2 \cdot sn^2(\omega)} \qquad 6.6$$

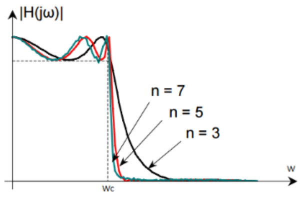


Figure 6.6 Elliptic filter characteristic.

6.4 SubVIs and express VIs

6.4.1 SubVIs

All VIs that we create ca be called from other virtual instruments. These contain Block Diagram, as well as the Front Panel. The appearance of a subVI is given by its **icon**. After creating the Block Diagram and the Front Panel of a VI, we will create an icon and its associated connector panel so that we can use this created VI as a sub-VI within other Block Diagrams. Every VI displays in the right corner of the Front Panel and Block Diagram an icon. For individualization or its editing double-click is used with the left mouse on the icon in question.

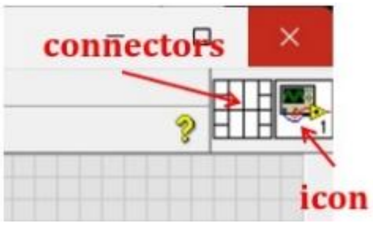


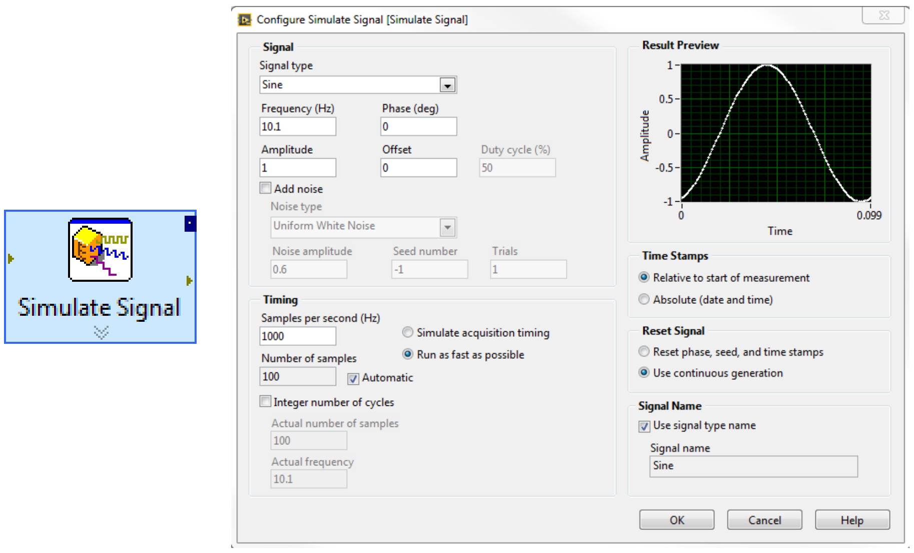
Figure 6.7 Connectors panel and icon.

To use the VI as a subVI, building a connector panel is also required. Connector Panel is a set of terminals that correspond to the control elements and indicators of that VI and can be seen in the right corner of the Front Panel next to the icon. This connector panel defines the inputs and outputs that you can use for the subVI.

Each rectangle represents a connector panel terminal. We will use these rectangles to assign the terminal input or output. We select the desired control element or indicator from the Front Panel, then click on one of the rectangles in the connector panel to associate a terminal.

6.4.2 Express VIs

Express VIs are designed to complement the joint operations commonly used in the acquisition, analysis and presentation of data. The difference between an Express VI and a subVI is that the user does not have access to the Block Diagram. The configuration of an Express VI is done hen placing it on the Block Diagram, when a configuration window is opened automatically.



Signal

Signal type: Sine

Frequency (Hz): 10.1 | Phase (deg): 0

Amplitude: 1 | Offset: 0 | Duty cycle (%): 50

☐ Add noise

Noise type: Uniform White Noise

Noise amplitude: 0.6 | Seed number: -1 | Trials: 1

Timing

Samples per second (Hz): 1000

○ Simulate acquisition timing
● Run as fast as possible

Number of samples: 100 ☑ Automatic

☐ Integer number of cycles

Actual number of samples: 100

Actual frequency: 10.1

Time Stamps

● Relative to start of measurement
○ Absolute (date and time)

Reset Signal

○ Reset phase, seed, and time stamps
● Use continuous generation

Signal Name

☑ Use signal type name

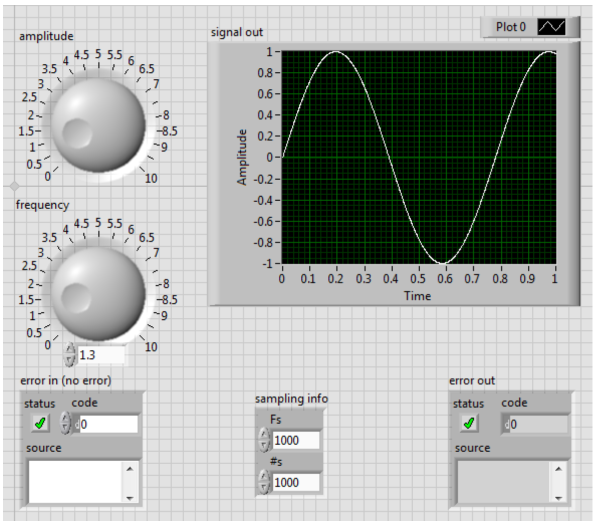
Signal name: Sine

Figure 6.8 The appearance of an Express VI and its configuration window.

6.5 Applications

6.5.1 Example 1 – signal generation

In this application, a method for generating a sinusoidal waveform is implemented.



frequency: 1.3

error in (no error): status ✓, code 0, source

sampling info: Fs 1000, #s 1000

error out: status ✓, code 0, source

Figure 6.9 Front Panel of the first example.

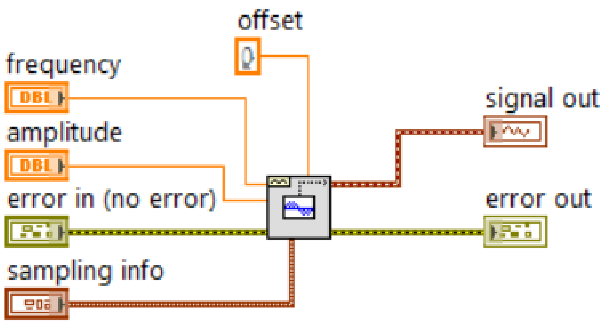


Figure 6.10 Block Diagram of the first example.

After making the Block Diagram and Front Panel, follow the steps for creating the subVI, shown in paragraph 6.4.

6.5.2 Example 2 – noise generation

For noise signal generation, the application below is proposed. The Block Diagram uses the CASE structure to be able to select different types of noise.

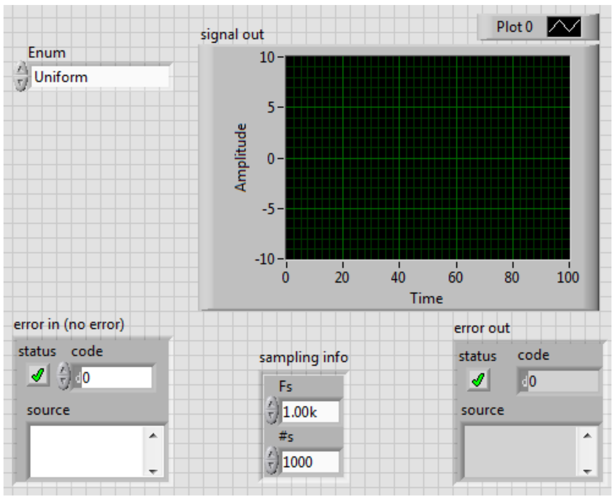


Figure 6.11 Front Panel for the second example

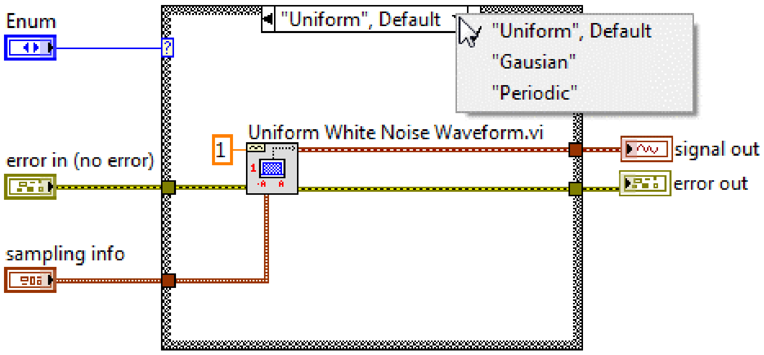


Figure 6.12 Block Diagram for the second example

After making the Block Diagram and Front Panel, follow the steps for creating the subVI, shown in paragraph 6.4.

6.5.3 Example 3 – digital filter implementation

The Block Diagram uses the CASE structure to be able to select different types of filters.

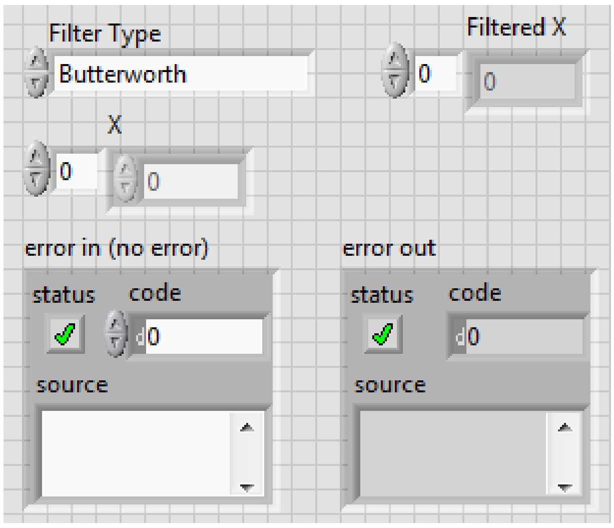


Figure 6.13 Front Panel for the third example.

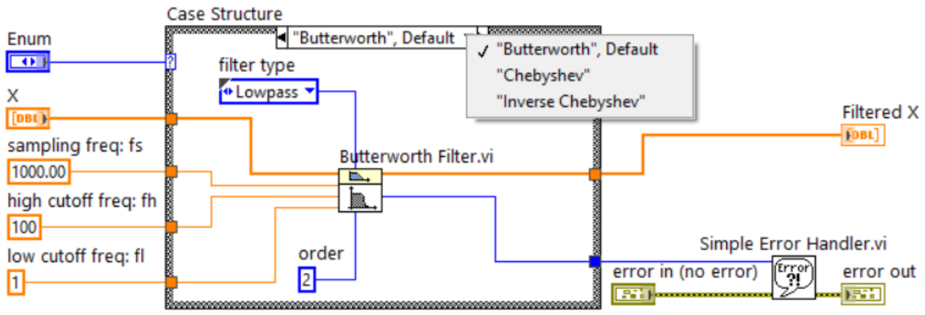


Figure 6.14 Block Diagram for the third example.

After making the Block Diagram and Front Panel, follow the steps for creating the subVI, shown in paragraph 6.4.

6.6 Questions and exercises

1. Improve the functionality of the first example, by using an enum button to select between different types of signal generation: sine/ rectangular/saw tooth/ triangular.
2. Develop a main VI which integrates all the previously created subVIs.

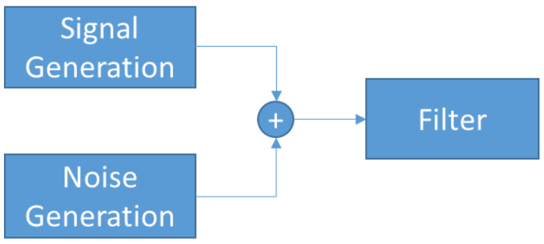


Figure 6.15 main.vi structure.

7 Debugging and Optimizing VIs

Very often, when creating Vis, at the end of the design phase, we may encounter issues and errors.

7.1 Error correction

VI Debugging is carried out automatically by the integrated compiler, which, during the application building, checks constantly for semantic and syntax errors.

When the Run button is not broken, the VI can be compiled and executed. If an error in the program exists, the VI cannot be run.

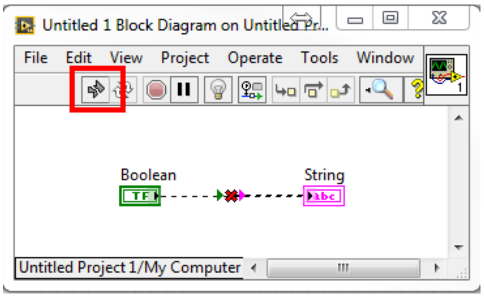
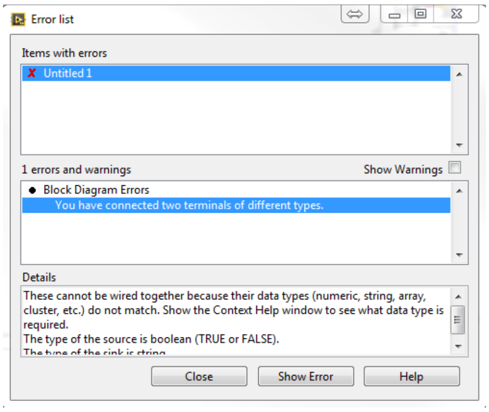

Untitled 1 Block Diagram on Untitled Pr...
File
Edit
View
Project
Operate
Tools
Window
Boolean
String
Untitled Project 1/My Computer

Figure 7.1 A broken Run VI button.

To run the application, the errors found in the **Errors List** need to be corrected.



Block Diagram Errors
You have connected two terminals of different types.

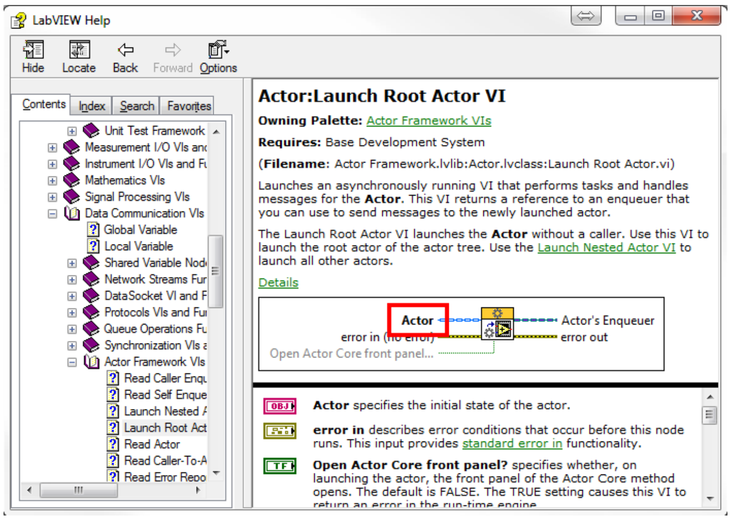
Details

These cannot be wired together because their data types (numeric, string, array, cluster, etc.) do not match. Show the Context Help window to see what data type is required.
The type of the source is boolean (TRUE or FALSE).

Figure 7.2 Error List Window

The most usual causes of a broke Run button are:

- Broken connections in the diagram, for example:
    - Connecting a Boolean control to a string indicator,
    - Connecting two numeric controls together.
- A terminal required to be connected is not connected. These types of terminals, for LabVIEW predefined functions, are represented in **BOLD** in the Help Window. Such an example is present in Figure 7.3.
- An error exists in a subVI.



Actor:Launch Root Actor VI

Owning Palette: Actor Framework VIs

Requires: Base Development System

(**Filename**: Actor Framework.lvlib:Actor.lvclass:Launch Root Actor.vi)

Launches an asynchronously running VI that performs tasks and handles messages for the **Actor**. This VI returns a reference to an enqueuer that you can use to send messages to the newly launched actor.

The Launch Root Actor VI launches the **Actor** without a caller. Use this VI to launch the root actor of the actor tree. Use the Launch Nested Actor VI to launch all other actors.

Details

Actor specifies the initial state of the actor.

error in describes error conditions that occur before this node runs. This input provides standard error in functionality.

Open Actor Core front panel? specifies whether, on launching the actor, the front panel of the Actor Core method opens. The default is FALSE. The TRUE setting causes this VI to return an error in the run-time engine

Figure 7.3 Help Window, with required terminals.

7.2 Debug techniques

Debug techniques are used when there are unexpected results at the output of a VI and NOT when the run button is broken (which indicates error presence).

In case of unexpected results or behavior, the following situations must be verified:

- If unconnected subVIs exist,
- If initial data are correct,
- If undefined data appears,
- If data representation is correct,
- If nodes are executed in the correct order.

There are four ways of debugging, discussed next.

Execution Highlighting – tool used for following or visualizing values of the data flow on each wire. This does not execute the application in real time, it slows down significantly the running of the application, to highlight the data values on all wires.

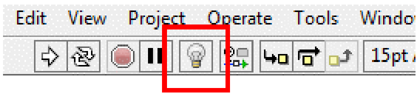


Figure 7.4 Execution Highlighting.

Single-Stepping – used for seeing the action of each function or subVI. These functions allow suspending the execution of a subVI and controlling execution time or returning to the initial state.

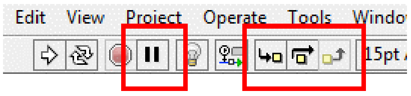


Figure 7.5 Single-Stepping.

Probes – used for immediate viewing of the values that are passing through wires and error checking.

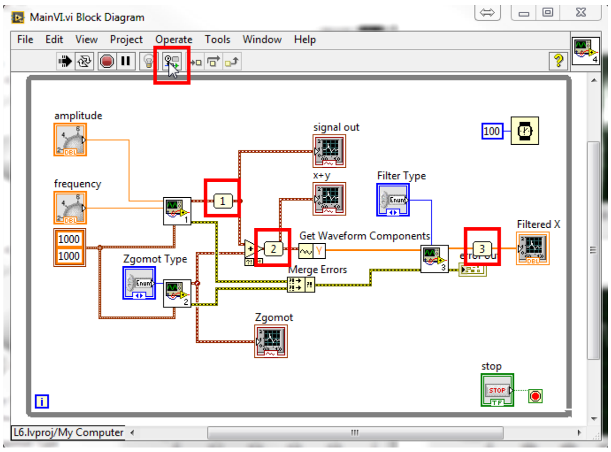
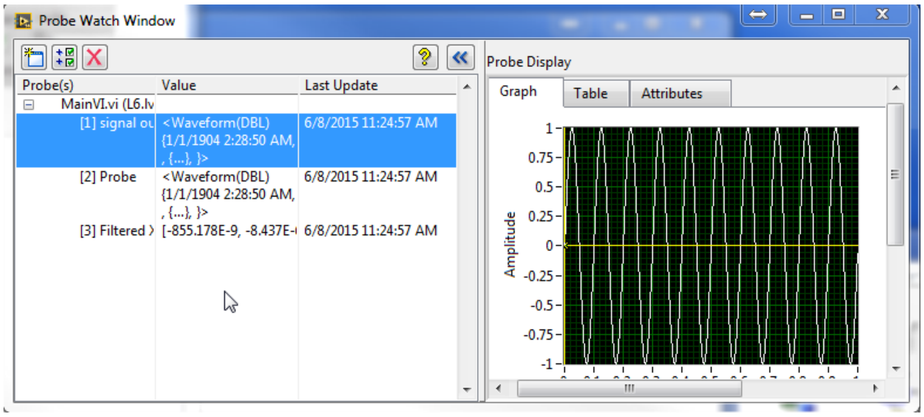


Figure 7.6 Probe.

Specific to this function is retaining values on the selected wires to allow viewing them at the end of the application.



Probe(s)	Value	Last Update
MainVI.vi (L6.lv		
[1] signal ou	<Waveform(DBL) {1/1/1904 2:28:50 AM, , {...}, }>	6/8/2015 11:24:57 AM
[2] Probe	<Waveform(DBL) {1/1/1904 2:28:50 AM, , {...}, }>	6/8/2015 11:24:57 AM
[3] Filtered)	[-855.178E-9, -8.437E-	6/8/2015 11:24:57 AM

Figure 7.7 Probe watch window.

Breakpoints – When the block to which a breakpoint was set is reached, the VI will pause, the **Pause** button will become red. The following actions can be performed:

- Single-Stepping,
- Probes,
- Modifying controls in the Front Panel,
- Pressing Pause for continuing the application.

7.3 Error handling

It is well known that every problem that may appear cannot be predicted. Without a mechanism to check the errors, we will know only that the VI does not work properly.

Error handling methods tell us how and why the errors occur. In LabVIEW there are two methods implemented:

- Automatic error handling method,
- Manual error handling method.

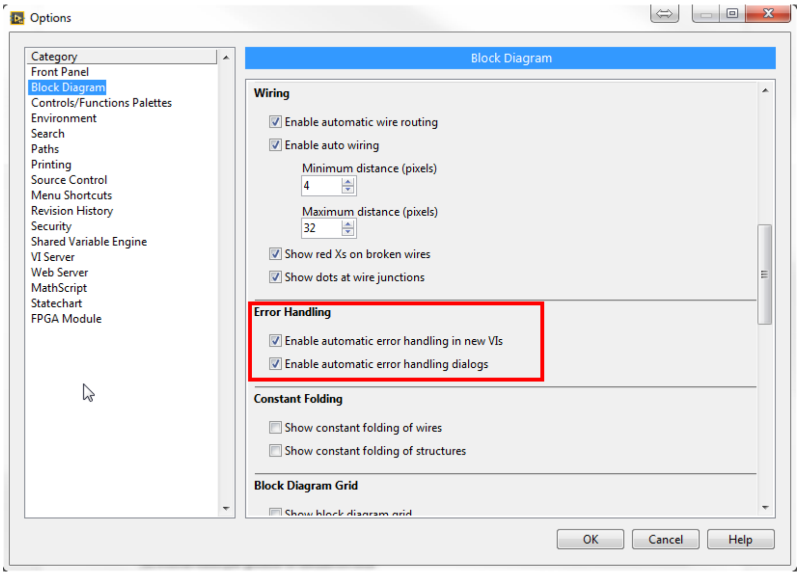
7.3.1 Automatic error handling

LabVIEW does automatic error checking. It takes the following actions when an error occurs:

- Suspends the execution of the program,

- Highlights the subVI or function that generated the error,
- Generates a list of errors.

This function can be disabled in the **Tools >> Options**, as can be seen in the figure below, marked by unchecking the two highlighted fields.



Options

Category
Front Panel
Block Diagram
Controls/Functions Palettes
Environment
Search
Paths
Printing
Source Control
Menu Shortcuts
Revision History
Security
Shared Variable Engine
VI Server
Web Server
MathScript
Statechart
FPGA Module

Block Diagram

Wiring
- [✓] Enable automatic wire routing
- [✓] Enable auto wiring

Minimum distance (pixels): 4

Maximum distance (pixels): 32

- [✓] Show red Xs on broken wires
- [✓] Show dots at wire junctions

Error Handling
- [✓] Enable automatic error handling in new VIs
- [✓] Enable automatic error handling dialogs

Constant Folding
- [ ] Show constant folding of wires
- [ ] Show constant folding of structures

Block Diagram Grid
- [ ] Show block diagram grid

OK | Cancel | Help

Figure 7.8 Disabling automatic error handling.

This is acceptable in the case of prototype applications, but it is not recommended for professional applications. This automatic error checking is not included in executable applications.

7.3.2 Manual error handling

This method allows the user to control the timing of appearance for error dialogs, errors propagate from the **error out** terminal of a function to the **error in** terminal of the next, ending at the Simple Error Handler VI function.

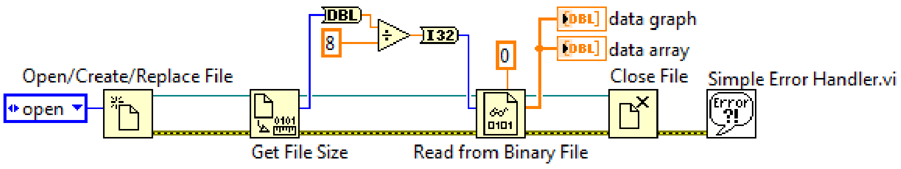


Figure 7.9 Manual error handling.

This method of routing the error path involves the appearance of a new error generation function for the situation where we have many parallel lines. This function is called **Merge Errors,** and it is important to know that it does not concatenate the errors. It will return an error **when the first error is encountered**. If it finds no errors, it will return the first warning.

To create **error in** and **error out** pins in a subVI, use error cluster controls and indicators. The error cluster, regardless of its type (control or indicator) has the following components:

- **Status** – returns TRUE when an error occurs. In case of warning or no error, it returns FALSE,
- **Code** – is the LabVIEW numerical identifier of the error or warning,
- **Source** – ID of the function in which the error occurs.

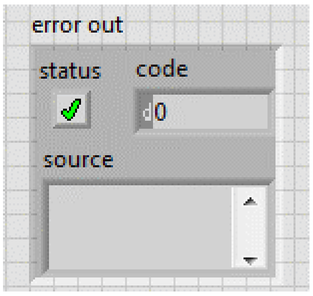


Figure 7.10 Error cluster with FALSE status (no error).

7.4 Applications

7.4.1 Example 1 – debug methods

This example proposes testing the four debugging methods, as well as introducing an error to observe the behavior in the error situations.

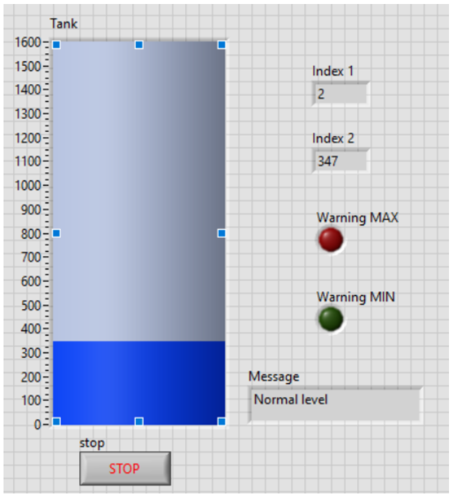


Figure 7.11 Front Panel of the first example.

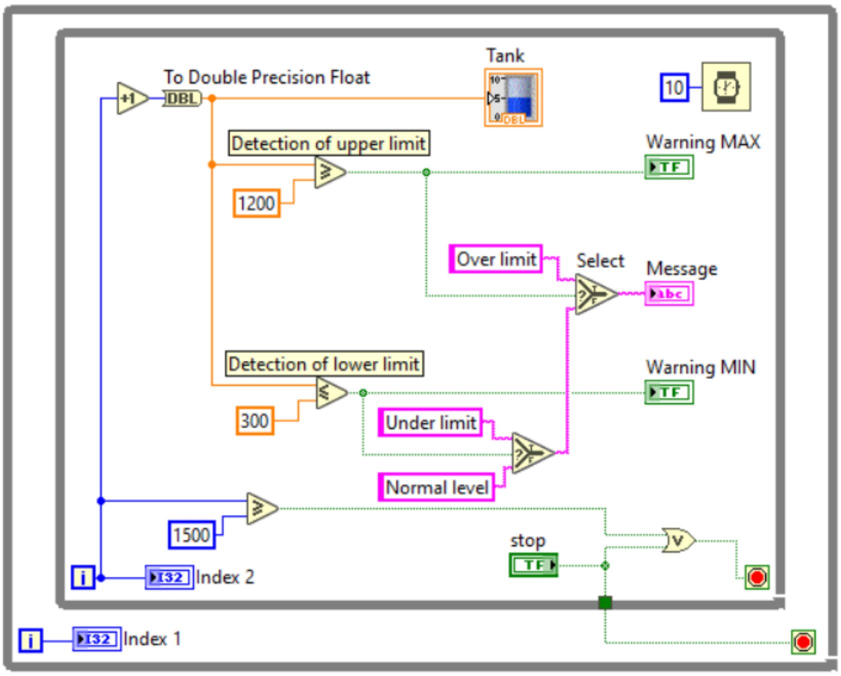


Figure 7.12 Block Diagram of the first example.

7.4.2 Example 2 – manual error handling

This application highlights an example of manual error handling.

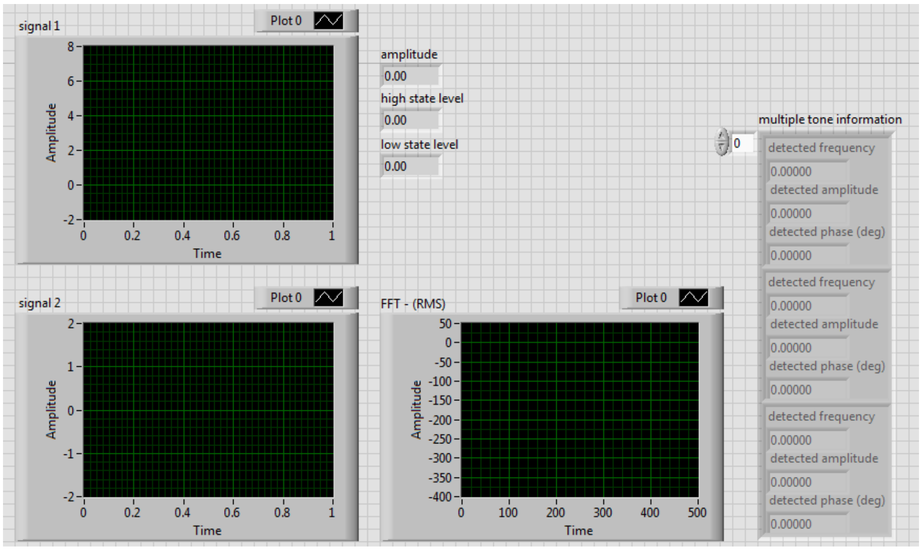


Figure 7.13 Front Panel of the second example.

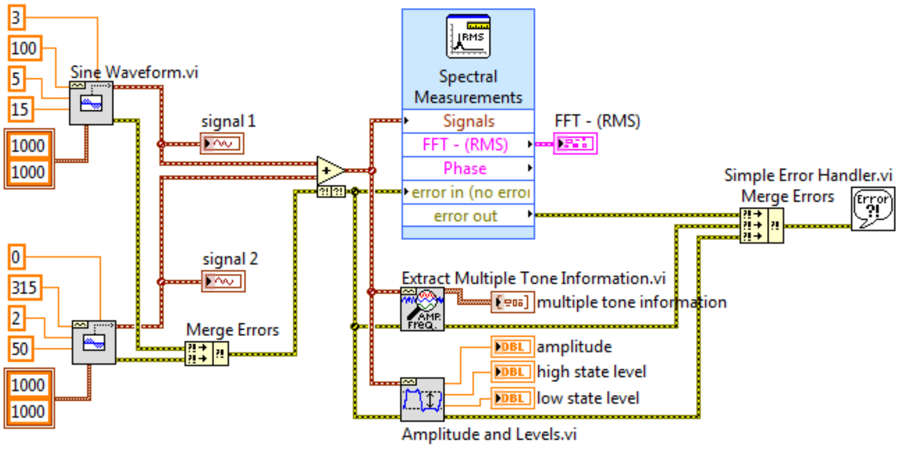


Figure 7.14 Block Diagram of the second example.

7.4.3 Example 3 – sound card data acquisition

With this application we highlight an example of manual error handling for the acquisition of a beep sound on the sound card of your computer.

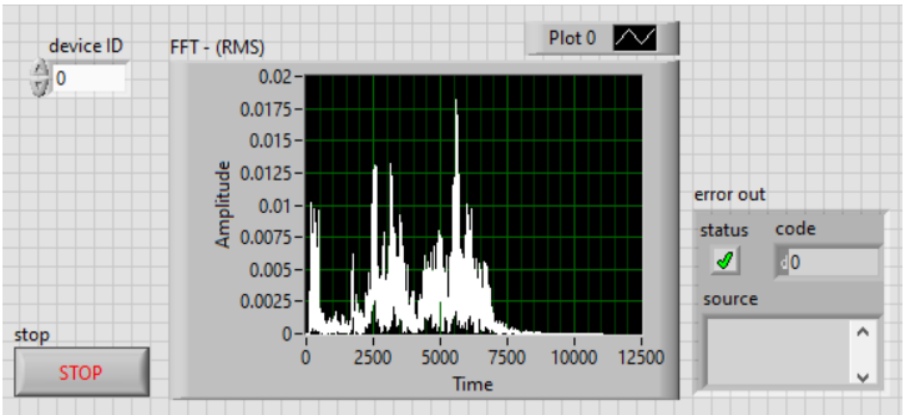


Figure 7.15 Front Panel of the third example.

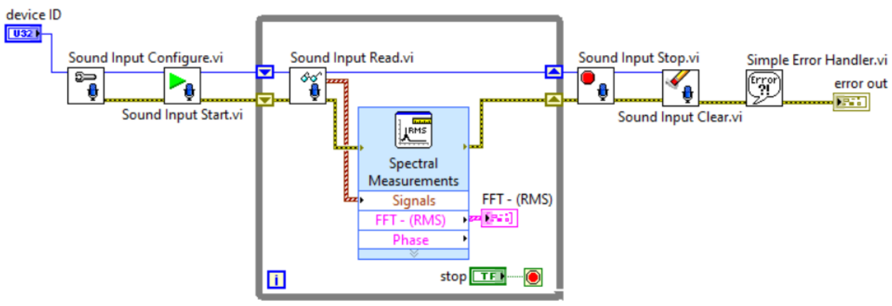


Figure 7.16 Block Diagram of the third example.

7.5 Questions and exercises

1. In Example 1, insert an error and run the error handling both enabled and disabled. What do you notice?
2. In the case of Example 2, what can you say about the function Merge Errors? Which module executes first between: Spectral Measurements, Extract Multiple Tone Information and Amplitude and Levels. Change the application so that the order of execution is the user's choice.

8 Design Patterns

This chapter covers code implementations and techniques that are solutions to specific problems in LabVIEW design.

8.1 Introduction

Many of the VIs we build perform sequential tasks. By default, LabVIEW is set so that no sequential programming is performed.

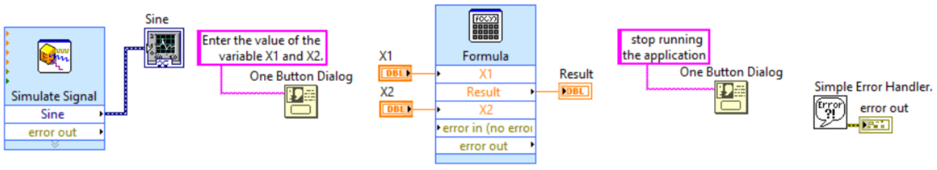


Figure 8.1 LabVIEW application without setting the execution order of the modules.

In the example above we can see the lack of sequential programming. If we run the application, the functions will be executed randomly, without a specific execution order, any of which can be executed first.

To impose an order execution of operations, it is recommended to use error clusters and refnums, which are references numbers of specific data types.

Not all features available in LabVIEW have error clusters available, such as the **One Button Dialog** function, also used in the example above.

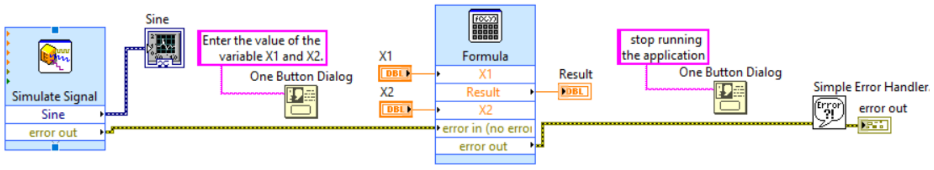


Figure 8.2 LabVIEW application and the use of existing error clusters.

One of the methods of choosing and fixing the execution order, in the absence of error clusters, is the use of the sequential structure. This has already been studied in Chapter 3.

The best way to create this VI is, however, to use error structures for modules that do not include error clusters. This structure is not directly available in LabVIEW but is created using the Case Structure, by routing an error cluster to the Case Selector.

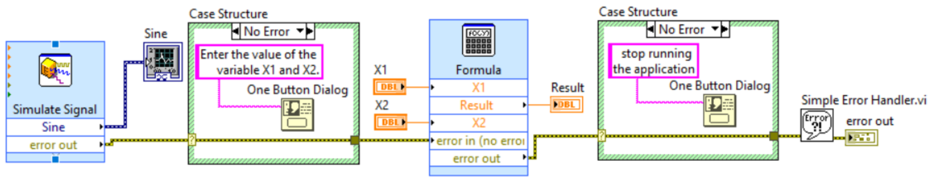


Figure 8.3 LabVIEW application using **Error Case Structure.**

Another problem is that this application will run once, the program stops running at the end of the last block execution.

Basically, this is the simple structure of a VI. It can perform a simple measurement, calculation or even display a result and does not require a user start or stop action.

A general VI has a more complex structure. As mentioned in a previous chapter, a general VI has three components:

- A start or an initialization,
- The code itself, which is typically contained in a While loop which helps to run continuously the same portion of code (multiple executions),
- One stop, used to free up resources.

Such a deployment technique is also used for systems that use state diagrams, but in their case, within the While loop, several actions can be implemented - during one iteration, only one action can run.

8.2 State programming

State programming helps us solve the following issues, which cannot be solved in sequential programming:

– If it is necessary to change the sequence execution order,

- If repetition of a sequence is necessary, more times than of another sequence,
- If some elements in a sequence are executed only when a certain condition is encountered,
- If it is necessary to stop the program immediately before the end of the sequence execution.

The State Transition Diagram is a flowchart that indicates the state of the program and transitions between states.

State – part of the program that meets a certain condition, performs an action, or awaits an event.

Transition – condition, action, or event that causes the transition from one sequence to another.

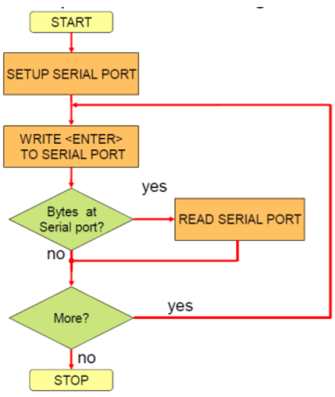


Figure 8.4 State Transition Diagram example.

8.3 State machines

State machine diagrams are physically implemented through state machines.

The most common uses are:

- For creating interfaces, where different user actions cause program sequence changes,
- For the testing process, where a state represents each segment of the process.

The state machine consists of a set of states and transition functions. Each state can lead to one or more states, or to the end of the process.

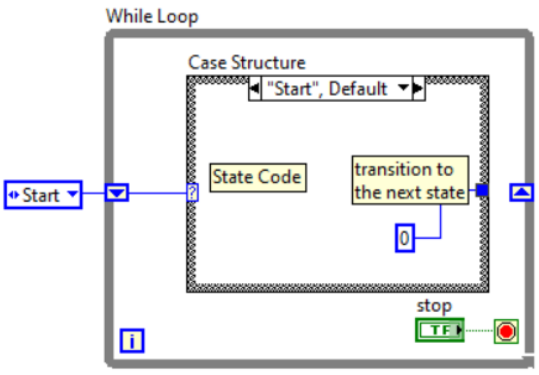


Figure 8.5 The basic structure of a state machine.

8.4 Event programming

Event – an asynchronous notification of an action that has taken place.

Asynchronous refers to the fact that a function starts an operation and may be recalled before the operation execution is complete. Events come from the user interface, externally, or from other parts of the program.

Interface events can be mouse actions, pressing a key, etc. External events can be timers or triggers that signal the end of a signal acquisition.

It is good to know that a control element on the Front Panel is a source of events. An event is the action on the event source. For example, changing the value (Value Change type) of the control element is an event.

Do not confuse the event with what it can do, or the role that control plays. The action of the control element is called a method.

Event programming - a programming method in which the program expects an event to occur before performing one or more functions.

By comparing classical programming with event programming, the following conclusions can be drawn:

The classic method involves running a continuous code snippet to check for any changes. This requires holding busy the CPU resource. They may not detect successive changes if they are running very fast.

Through the event programming method, we have the following benefits:

– the successive inspection is deleted,
– processor demand is reduced,
– the Block Diagram is simplified,
– the detection of all occurring events is guaranteed.

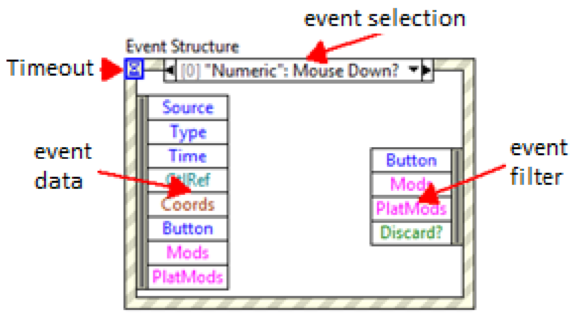


Figure 8.6 Event Detection structure.

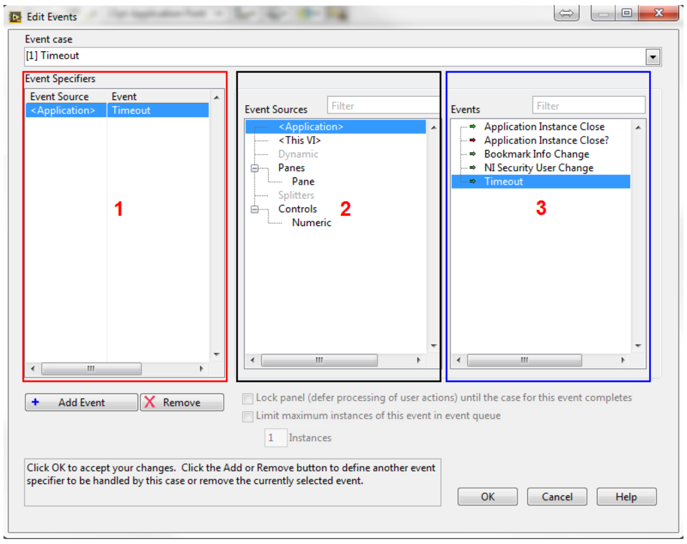
Timeout - Specifies the waiting time of an event (ms). If a value is specified, then Timeout case has to be created.

Event Data – Identifies data provided by LabVIEW when an event occurs. Similar to **Unbundle by Name**.

Event Filter – Identifies the data subgroup available in the Event Data that the structure can modify.

As functionality, it is recommended to place the Case structure in a While loop. Each event will be executed in an iteration of the While loop. The event structure is put into Sleep Mode when no events occur.

To set up each event, use the right mouse button and select **Edit Events Handled by This Case.**



Event case
[1] Timeout

Event Specifiers

Event Source	Event
<Application>	Timeout

Event Sources
Filter
<Application>
<This VI>
Dynamic
Panes
Pane
Splitters
Controls
Numeric

Events
Filter
Application Instance Close
Application Instance Close?
Bookmark Info Change
NI Security User Change
Timeout

Lock panel (defer processing of user actions) until the case for this event completes
Limit maximum instances of this event in event queue
1 Instances

Click OK to accept your changes. Click the Add or Remove button to define another event specifier to be handled by this case or remove the currently selected event.

Figure 8.7 Event Configuration Window.

In pane 1, a list of events is presented where the current case can be dealt with, in pane 2 the user can choose the source of the event, and in pane 3 the individual events of each source of event generation are presented.

The green arrow indicates that the event occurred, and LabVIEW processed it, and the red arrow indicates that the event occurred, but LabVIEW has not yet processed it.

8.5 Applications

8.5.1 Example 1 – state machine example

The first example refers to the creation of a State Machine. It starts from the Start state and continues, generating, in turn, the sinusoidal, rectangular and triangular signals each corresponding to cases 1, 2 and 3 respectively. From the last case, the sinusoidal case state will be called (case 1), without passing through the start state.

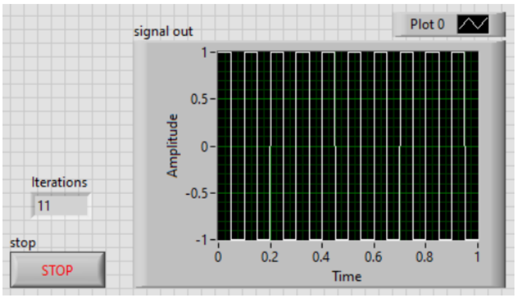


Figure 8.8 Front Panel of example 1.

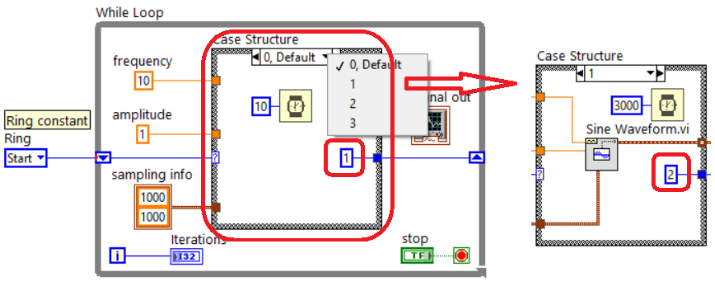


Figure 8.9 Block Diagram of example 1.

8.5.2 Example 2 - implementing the pooling method

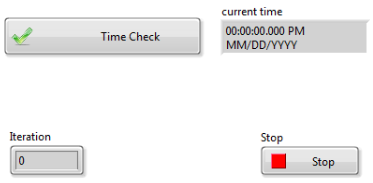


Figure 8.10 Front Panel of example 2.

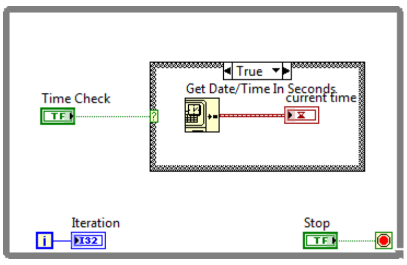


Figure 8.11 Block Diagram of example 2.

8.5.3 Example 3 - implementing an application using triggering events

An Event structure can be found in the **Structure Palette -> Event Structure.** To configure the event, after placing the structure in the Block Diagram, right-click and select **Edit Events Handled by This Case.**

In the window of the Figure 8.7, the Source (in Event Sources) and Event Type (in Events) will be selected. These will always appear at the top of the structure. To add a new event, use the right mouse button on the edge of the structure and select **Add Event Case**.

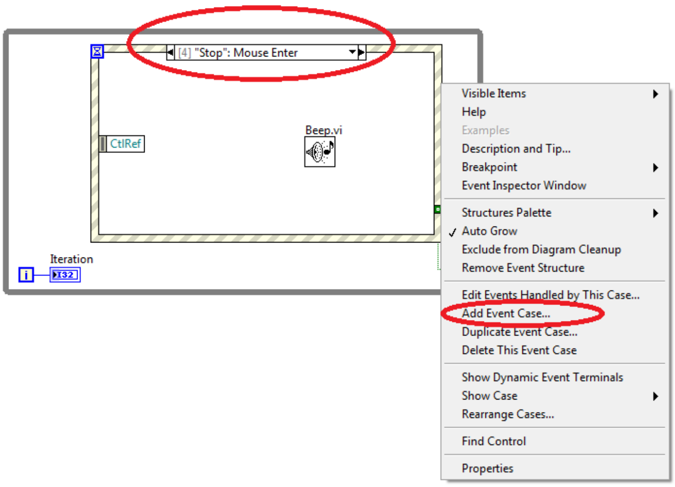


Figure 8.12 Adding a new event.

The event structure will be created so that it can process 5 events.

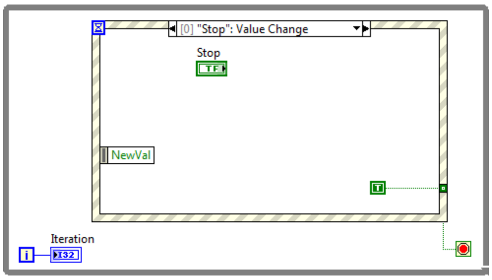


Figure 8.13 Case of event 0.

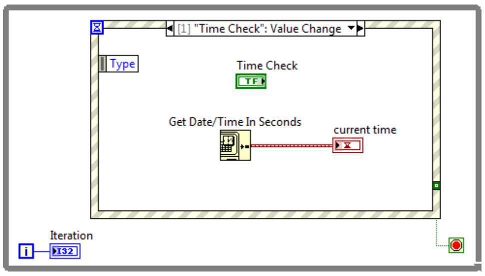


Figure 8.14 Case of event 1.

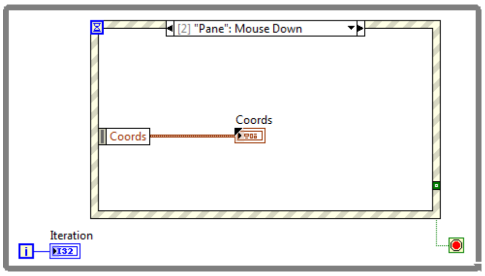


Figure 8.15 Case of event 2.

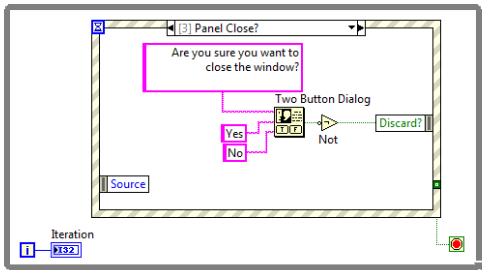


Figure 8.16 Case of event 3.

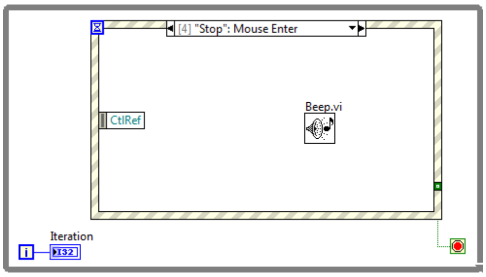


Figure 8.17 Case of event 4.

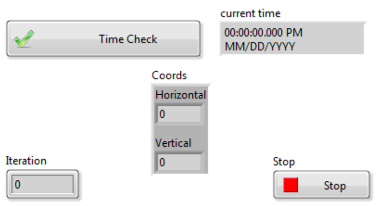


Figure 8.18 The Front Panel of example 3.

8.6 Questions and exercises

1. What is the advantage of using Events, compared to the application in example 2?

2. What is the functionality of each event in example 3?

9 Asynchronous Data Transmission and Data Synchronization

LabVIEW is a graphical programming language based on data sequence transmission (numerical values, strings, etc.). It is a dataflow language. This means that:

- Functions depend on data transmitted by other functions,
- Dependent functions do not execute until dependencies finish executing,
- Data is transmitted from one function to another via transmission lines.

However, in some cases, this method of transmission needs to be interrupted using asynchronous communication.

9.1 Asynchronous communication

Asynchronous communication refers to information transfer without using transmission lines. This communication method is used between:

- Parallel loops,
- Front panel and block diagram,
- VIs,
- Application instances (LabVIEW projects, executable files, etc.).

Information transmitted through this type of data communication is the actual data and notifications that an event has occurred.

This category includes **local variables**, already studied in previous applications, **notifiers**, **user events**, but also **queues**. One thing to know about local variables is that they are mostly used to update the value of a control or indicator and occasionally when reading a control or indicator variable. Notifiers and user events are not part of the current study.

9.2 Queues

Queues are used for data communication between parallel loops. They can hold a significant amount of data, based on the FIFO (first in, first out) method of data handling. They have the advantage of being able to hold any type of data.

The use of local variables to transmit data between parallel loops has some disadvantages:

- Duplicate reading of data is possible,
- Losing data is a possibility,
- One should create data write and read priorities (race conditions).

All the above-mentioned disadvantages can be avoided by using queues. The advantage of using a queue is that the producer and consumer will run as parallel processes, and their rates do not have to be identical. LabVIEW provides us with a whole palette of functions for the use of queues, and it is recommended to use this type of communication for the following types of applications:

- Communication between different sections of the same VI,
- Communication between different VIs.

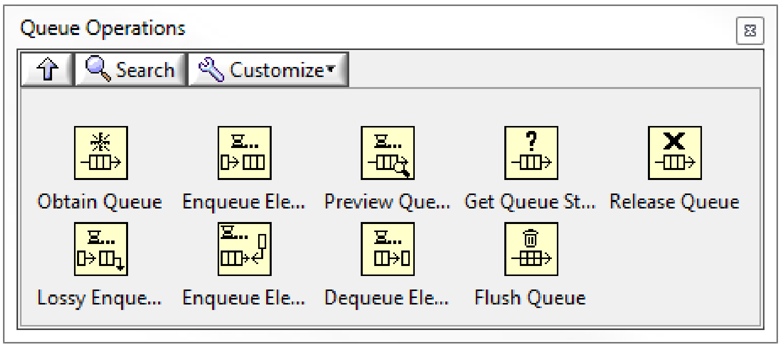


Figure 9.1 The functions palette implemented in LabVIEW for queues.

When using queues a design must be followed. The design implies using at least two parallel loops: one loop for creating the stack, adding one element at each iteration, and one loop for retrieving elements from the stack.

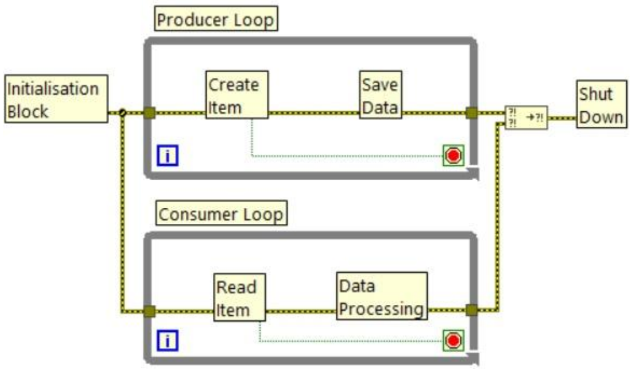


Figure 9.2 Producer/Consumer design.

Basically, this procedure separates the tasks of producing and consuming data with different speeds.

The stack is a temporary memory where data communicated between two devices or multiple loops is saved.

9.3 Data synchronization

Through execution synchronization, we provide the application with a function by which we give time to the processor to complete other tasks.

9.3.1 Execution synchronization

When we have a design where timing is based on the occurrence of an event, we don't need to determine exactly how often the executions are synchronized because the portion of the design executes when an event occurs.

One can see the example in Figure 9.3 is not using any timing function because it is directly integrated by the existence of the **event structure** and the **Dequeue element** function. The **Event Structure,** placed in the Producer loop, controls the execution of this loop. **Dequeue element** function within the Consumer loop, waits until an element appears in the stack, thereby controlling the execution of this loop. In conclusion, this application does not require synchronization because this is done via event triggering.

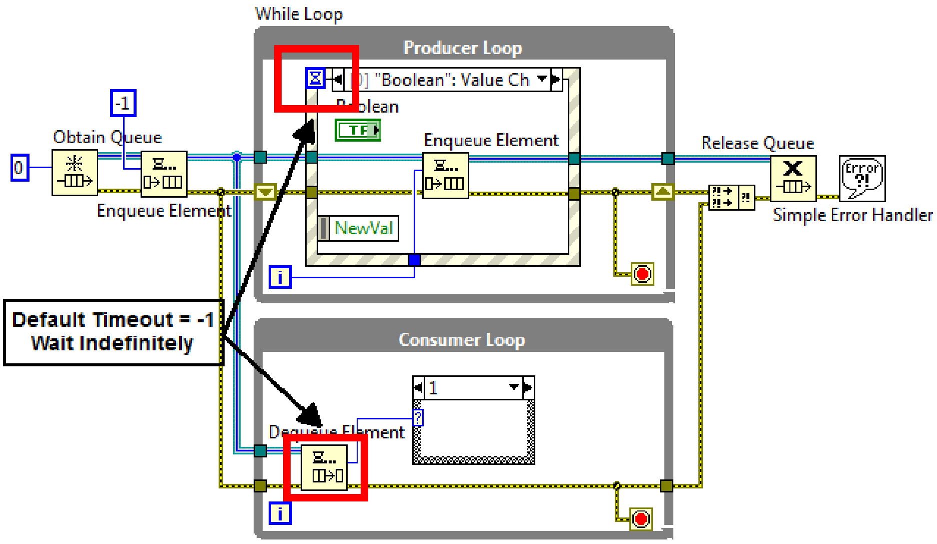


Figure 9.3 Highlighting the timeout.

9.3.2 Software synchronization

This type of synchronization must allow the continuous running of the design. **Wait** and **Wait Until Next ms Multiple** functions are used more for execution synchronization than for software synchronization.

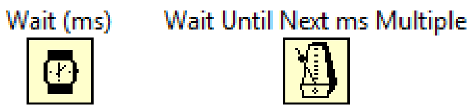


Figure 9.4 Wait and Wait Until Next ms Multiple functions.

We have already used these functions in a few applications made so far to achieve software synchronization, but while they are suitable for execution synchronization they are not preferred in this case.

An alternative is to use the timeout functionality associated with event structures and the use of stacks (Queues). This can be seen in Figure 9.5. In this example, even if no event is triggered or the stack has no elements, the loops continue to execute at regular time intervals.

Producer Loop executes every 100ms even if no event occurs. The existence of this constant causes the event structure to wake up from sleep mode and execute the implemented code in case of Timeout.

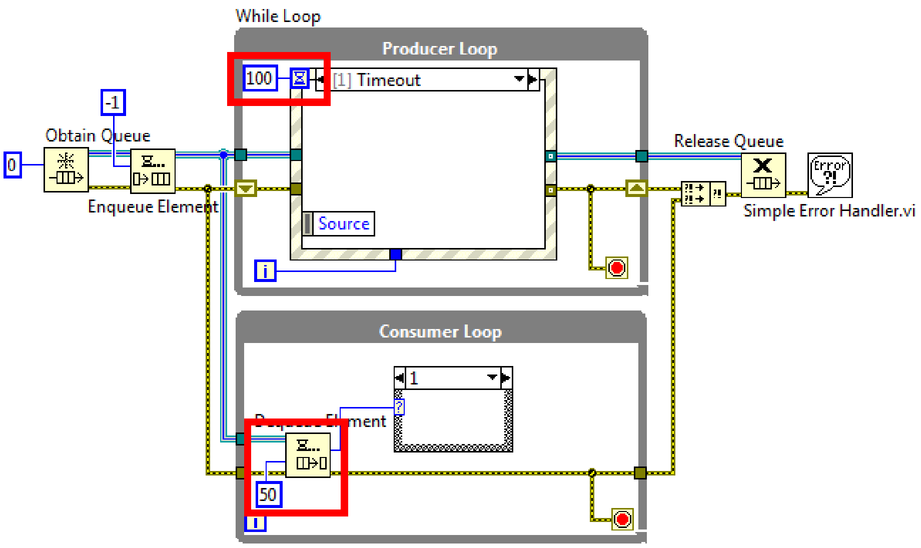


Figure 9.5 Setting the Timeout to real values in ms.

Consumer Loop is executed every 50ms even if there are no elements in the stack.

Synchronization, in this case, implies the execution of the implemented code, in the two loops, at specified time intervals.

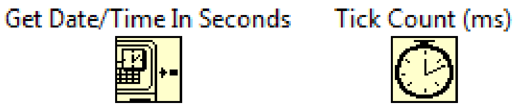


Figure 9.6 Get Date/Time In Seconds and Tick Count (ms) functions.

Get Date/Time In Seconds function returns a pattern for current date and time. Generally, it is used for comparing execution times. Also, it is useful for periodic measurements or actions for which function **Wait (ms)** can introduce delays.

LabVIEW provides the **Tick Count (ms)** function as well, which can be used for obtaining the relative time. This function is used for benchmarking code while **Get Date/Time In Seconds** is used for indefinite runs. This is because the **Tick Count (ms)** function can return values between 2^{32}-1 and 0.

9.4 Applications

9.4.1 Example 1 – implementing an application using the producer/consumer loop model.

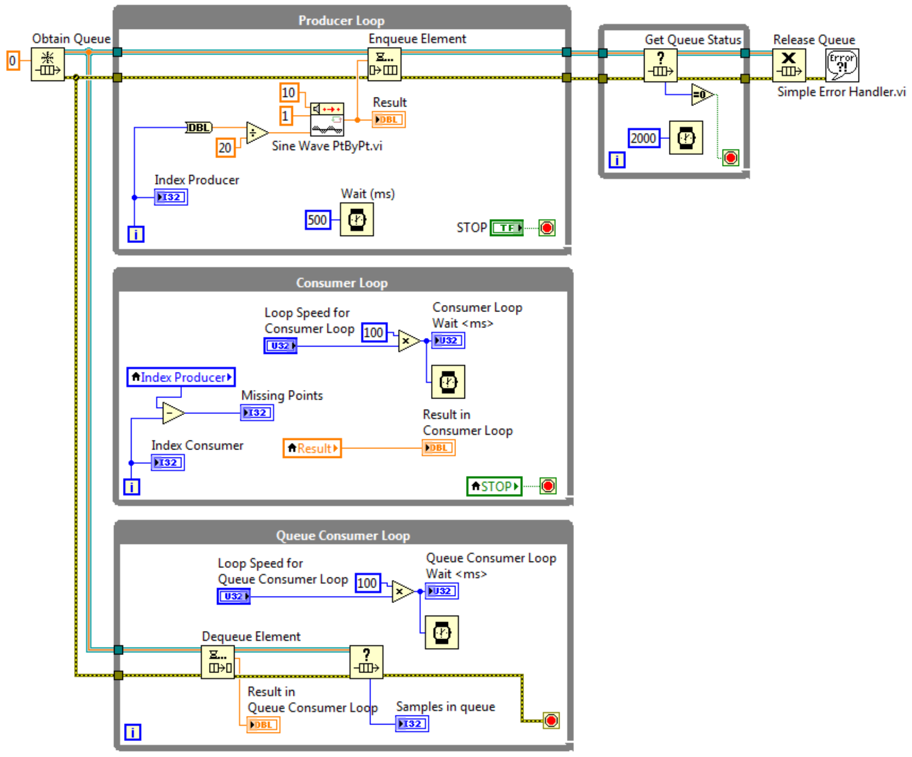


Figure 9.7 Block Diagram of the first example.

Within Producer Loop, a point-by-point sine signal is generated. The signal is comprised of 20 points per period. Each point is generated every 500 ms and saved on the stack.

In the "Consumer loop", using a local variable, the result of the signal generated in the "Producer loop" is displayed.

Within the "Queue Consumer Loop" data is extracted from the stack element by element and displayed on a graphic indicator.

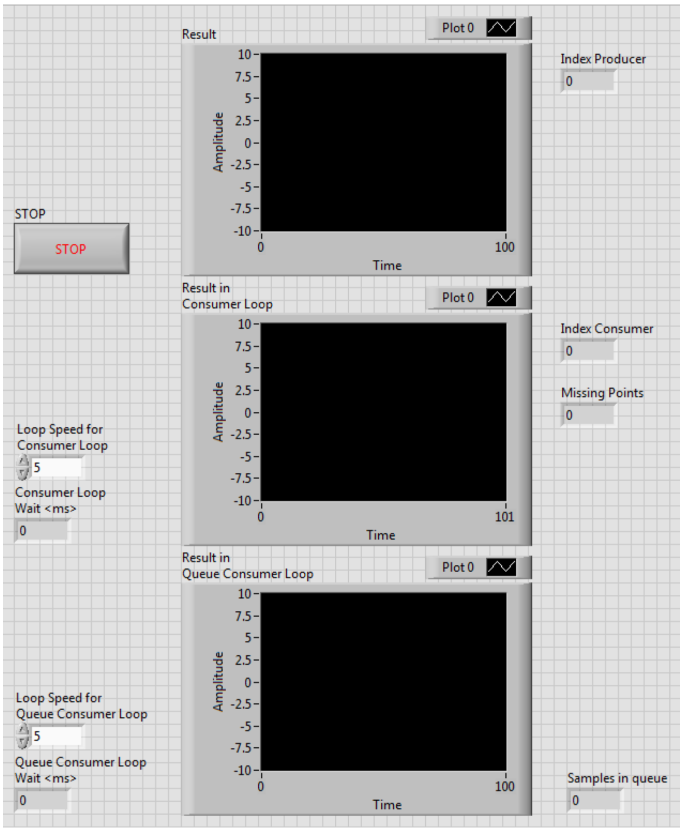


Figure 9.8 Front Panel of the first example.

9.4.2 Example 2 – using timing functions

This example shows two methods of measuring the time elapsed from the start to the end of the program.

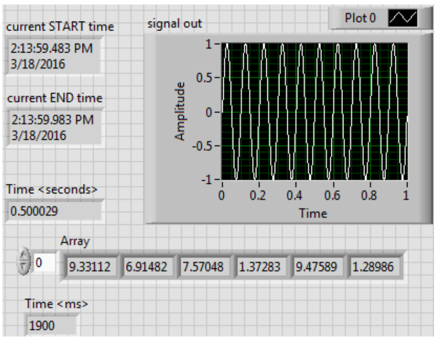


Figure 9.9 Front Panel of the second example.

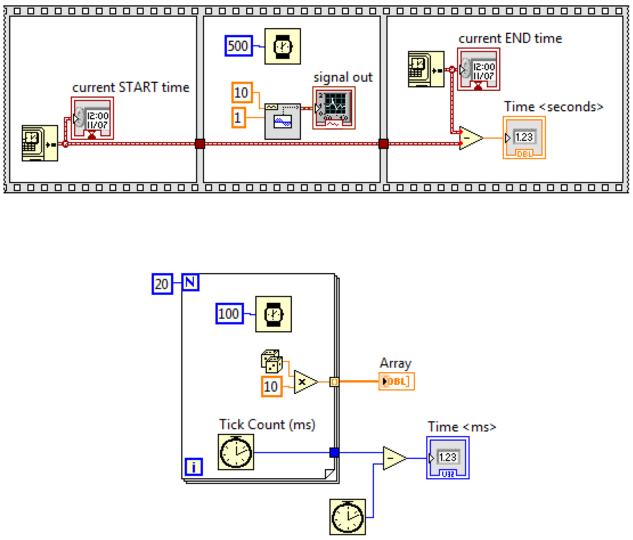


Figure 9.10 Block Diagram of the second example.

9.5 Questions and exercises

1. In the first example, what happens when the value of the "Loop Speed for Consumer Loop" control element is changed? But in the case of changing the value of the "Loop Speed for Queue Consumer Loop" control element?
2. In the first example, what do you notice when you run the application with the values of the control elements "Loop Speed for Queue Consumer Loop" and "Loop Speed for Consumer Loop" set to a high value (Ex: 20), and at some point, you want to stop running by pressing the STOP button?
3. Create an application to find the execution time of example 1.

10 Frequency Applications

10.1 Fourier analysis

The Fourier analysis consists in determining the coefficients of the Fourier series for a known x(t) signal.

Fourier synthesis consists in building an x(t) signal from a function sum $\{x_k(t)\}_k$ pondered weighted with a_k coefficients.

Harmonic Fourier series emphasizes the amplitude and phase of one component, having the order k. We consider the following trigonometric Fourier series:

$$x(t) = C_o + \sum_{k=1}^{\infty} [C_k \cdot \cos(k \cdot \omega_o \cdot t) + S_k \sin(k \cdot \omega_o \cdot t)] \qquad 10.1$$

If we consider a k-order term from the above sum, and apply it to the equations:

$$C_k = A_k \cdot \cos(\varphi_k);\ S_k = -A_k \cdot \sin(\varphi_k). \qquad 10.2$$

We obtain:

$$\begin{aligned} &C_k \cdot \cos(k \cdot \omega_o \cdot t) + S_k \sin(k \cdot \omega_o \cdot t) \\ &\quad = A_k \cdot \cos(k \cdot \omega_o \cdot t + \varphi_k) \end{aligned} \qquad 10.3$$

So, we can write the trigonometric Fourier series:

$$x(t) = \sum_{k=0}^{\infty} A_k \cdot \cos(k \cdot \omega_o \cdot t + \varphi_k) \qquad 10.4$$

Equation 9.4 represents the Harmonic Fourier series. The term of order k = 1 is the fundamental component, and the k-order is the harmonic k in the harmonic Fourier representation. The connection between coefficients of the Harmonic Fourier series and Trigonometric Fourier series results from equation 9.2:

$$\begin{cases} A_o = C_o \\ A_k = \sqrt{C_k^2 + S_k^2};\ \varphi_k = -\operatorname{arctg}\dfrac{S_k}{C_k} \end{cases} \qquad 10.5$$

Where A_k is the amplitude of k-order harmonic and φ_k is the k-order harmonic phase.

The Harmonic Fourier series emphasizes the amplitude and phase of a k-order harmonic.

The periodic signal – A signal is periodic if the function x(t), which describes it, is periodic. In this case:

$$x(t) = x(t + k \cdot T_S), k \in \mathrm{N} \qquad 10.6$$

T_s being the smallest time interval which satisfies the above equation, interval named the periodic signal's period (referring to the harmonic Fourier series).

Periodic signals are used in medical equipment, in telecommunication systems (voice and data compression, filtering, signal multiplexing), in industry (monitor and control processes).

Fourier analysis of periodic signals is called also harmonic analysis. Through the equations that describe the harmonic Fourier series, a connection is established between the time function and the harmonic group, of frequency $f = k*f_0$, $k \in N$.

In signals study, it is necessary to represent them in time domain (the waveform), as well as in frequency domain, which is the spectrum diagram.

If we start from a component with frequency $f = k*f_0$, this component will be defined with the appropriate amplitude:

$$A_k \cdot \cos(k \cdot \omega_o \cdot t + \varphi_k)\ , k \in \mathrm{N}, A_k \in \mathrm{R}, \varphi_k \in \mathrm{R} \qquad 10.7$$

10.2 Discrete Fourier Transform

As seen in Figure 9.1, Discrete Fourier Transform (DFT) transforms an input signal formed from N samples in two output signals formed from N/2+1 samples.

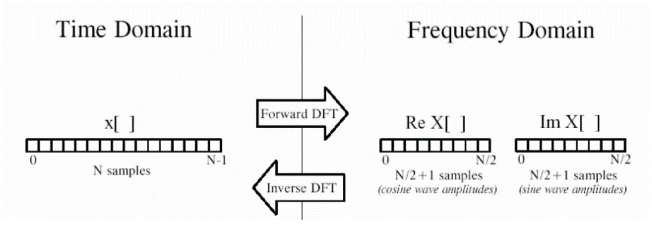


Figure 9.1 DFT

10.3 Applications

10.3.1 Example 1 – phase and amplitude representation

This example shows the representation of phase and amplitude for two sine signals.

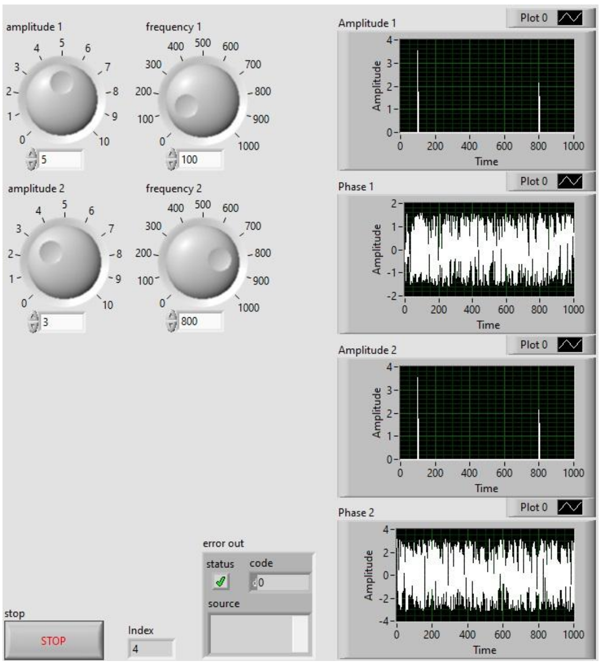


Figure 9.2 Front Panel of the first example.

For better viewing the spectrum, right click on the Waveform Graph and choose Visible Items -> Graph Palette.

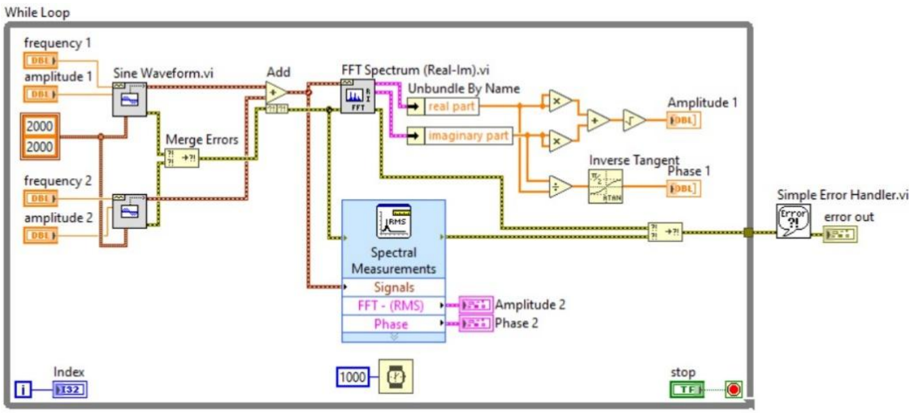


Figure 9.3 Block Diagram of the first example.

10.3.2 Example 2 - Discrete Fourier Transform

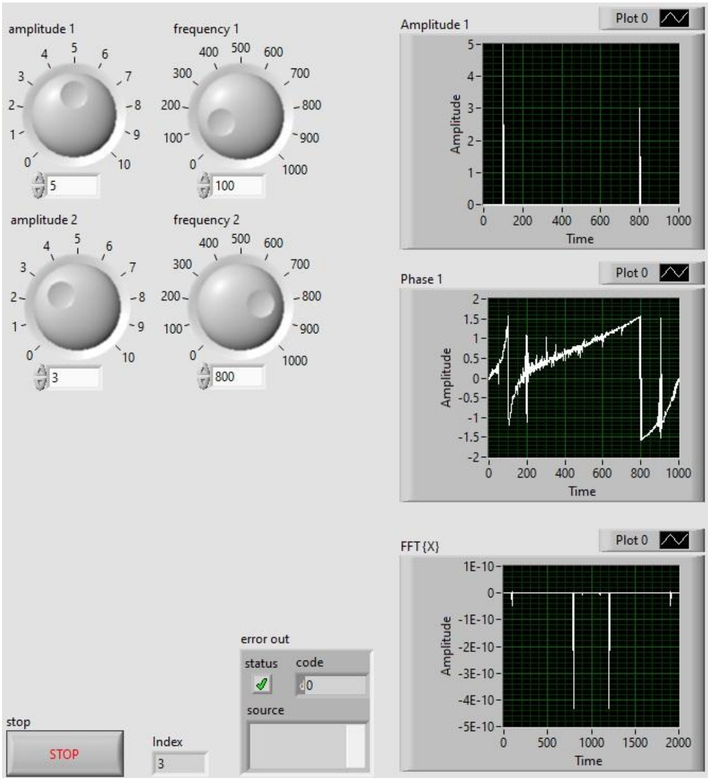


Figure 9.4 Front Panel of the second example.

Because the spectrum will be mirrored, we will display the data only for half the samples. It can be noticed that if the input signal has N samples, then the frequency spectrum will be in half. For that purpose, we will place, from the **Array Functions Palette**, **Split 1D Array**. At the index terminal we will route half the number of samples and we will extract the first subarray for the real part, and for imaginary part, respectively.

The resulted arrays after the split will be divided to the number of samples and multiplied by 2. After this operation, we will obtain the real part and imaginary part for the signal to which we applied DFT.

With two waveform graphs we will display the frequency and phase spectrums.

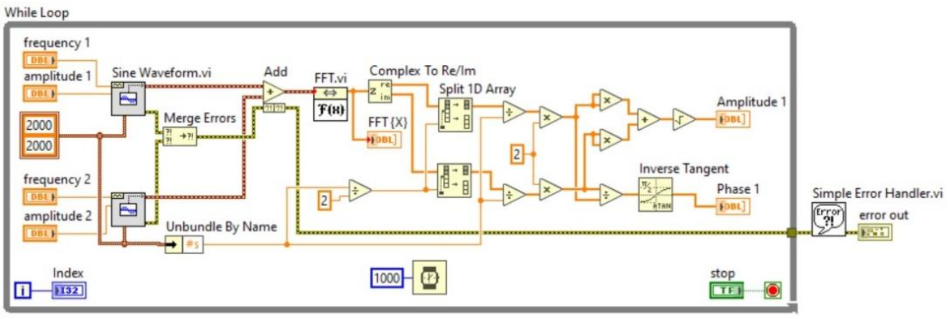


Figure 9.5 Block Diagram of the second example.

10.4 Questions and exercises

4. Modify the values of the input signals and observe what happens at the output.
5. Compare the spectrums obtained in the first and second example.

11 User Interface Control

User Interface Control can be done through several functions implemented in LabVIEW, such as:

- VI Server Architecture,
- Property Nodes,
- Invoke Nodes,
- Control Reference.

11.1 VI Server architecture

VI Server architecture is a collection of functions, properties and methods that allow programmable access to objects and functionalities in LabVIEW. Calling into LabVIEW and the VIs on a computer can be done remotely. Afterwards, they can be controlled by the implemented code. This architecture allows us to load and run the VIs dynamically.

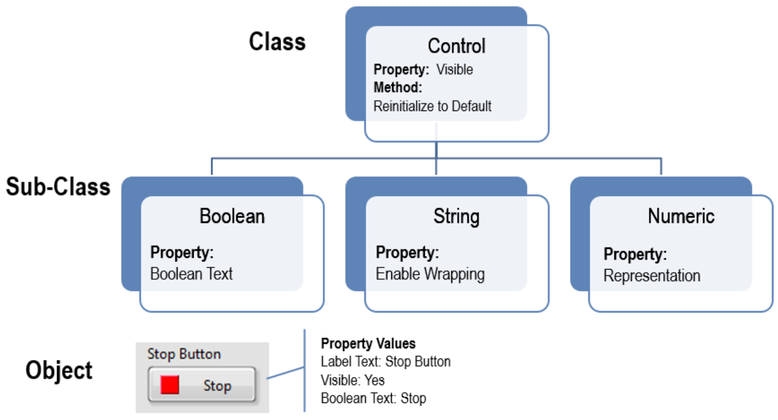


Figure 11.1 VI Server Hierarchy.

An **object** is an entity existent in the current application instance.

Properties are object attributes that can have the following functions: read/write, only read, or only write. Examples of the properties are color, position, dimensions, visibility, name, etc.

Methods are functions that operate on objects. The methods include reinitialization of the default values and the export of the graphical images.

VI Server has object-oriented architecture. Each object of the VI server is part of a class. The class determines which property, and method can be applied to the object. Also, subclasses of the control classes exist, subclasses in which objects are defined depending on their type, as seen in Figure **11.1**.

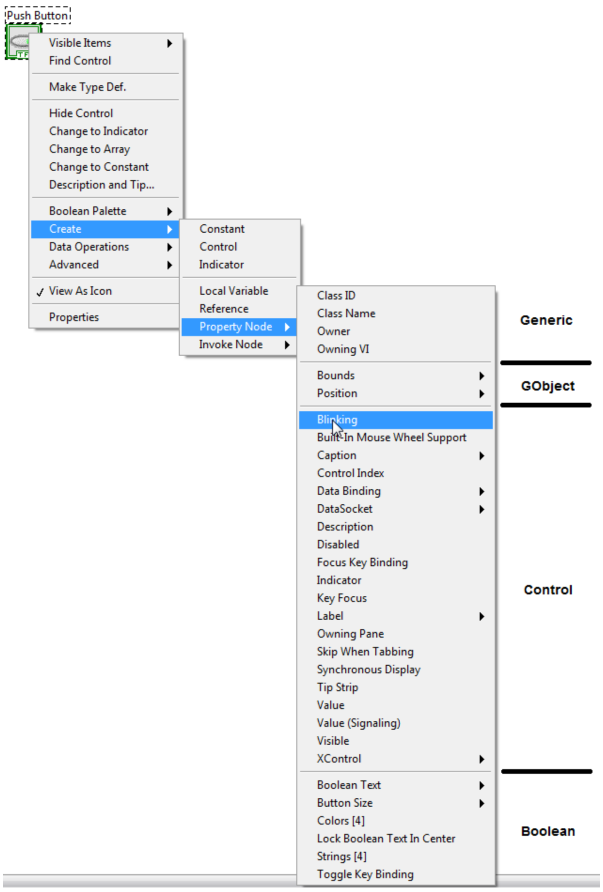


Figure 11.2 Front Panel of a VI.

In Figure **1.2**, the VI Server hierarchy can be observed. The groups Generic, GObject, Control and Boolean are classes. The Control class is a GObject subclass, which, in turn, is a subclass of the Generic class.

11.2 Property nodes

These types of functions offer the possibility of reading and writing (modifying) the properties of an object. Through property nodes, one can:

- Change the color of different (graphical),
- Disable and enable controls,
- Get the location of a control or indicator.

Property nodes allow the changing operations to be performed programmatically, and due to the existence of various properties for each object, it is recommended to obtain information about existing properties from the LabVIEW Context Help, before using property nodes.

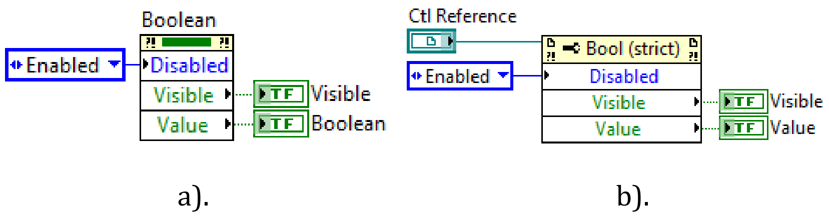


Figure 11.3 Property Nodes: a). Implicit, b). Explicit.

The execution is done from the properties on top to the ones on the bottom in the list. If an error appears, the property node execution will stop, returning the error. To overwrite the default behavior, it is recommended to access the optional menu via right click and select **Ignore Errors Inside Node.**

Among the most used properties are:

- **Position:** the position of the element on the panel, expressed in pixels horizontally and vertically,
- **Bounds:** the dimensions of the element, expressed in pixels,
- **Visible:** if this property has the value False, the element is no longer displayed in the panel (although it continues to exist and has a terminal in the diagram),

- **Disabled:** the value 0 means that the user can act on that element, the value 1 means that the element cannot be acted on (it is disabled), the value 2 means that the element is disabled and displayed in "clear" colors,
- **Blinking:** when this property has the value True, the element is displayed "blinking",
- **Label:** properties for formatting the text in the label.

11.3 Invoke nodes

Invoke nodes are used to access methods and actions performed on objects. Through **Invoke nodes**, one can obtain: VI version, printing, reinitialization to default values.

Most of the methods have several parameters, as is the case with property nodes; to obtain supplementary information about the methods, the usage of Context Help is recommended.

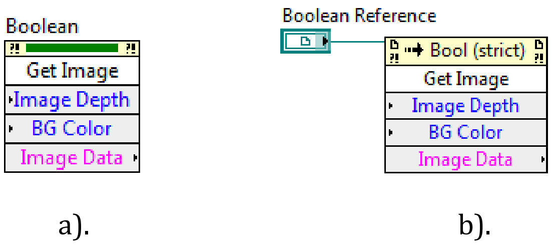


Figure 11.4 Invoke Nodes: a). Implicit, b). Explicit.

If the background of a parameter is grey, it is set as optional. The creation method for Property nodes and Invoke nodes is the same: accessing the menu via right click on the object one wishes to create this type of function and selecting **Create->Property Nodes** or **Create->Invoke Nodes.**

11.4 Control reference

A Property Node element created for a Front Panel object is an implicit property node, which is directly tied to the existent object. A generic property node, with a connected reference is an explicit property node. The latter is used especially when a generic property node is needed, which is part of a subVI.

Control reference is a reference to an object from the Front Panel. Control references connect the object to a generic property node and are used when passing references from the main VI to subVIs.

Control reference is created via right click on the object for which the reference is needed, selecting **Create->Reference**.

11.5 Applications

11.5.1 Example 1- property nodes usage

The example proposes programmatical modification of a level indicator color and setting the minimum and maximum limits of a knob.

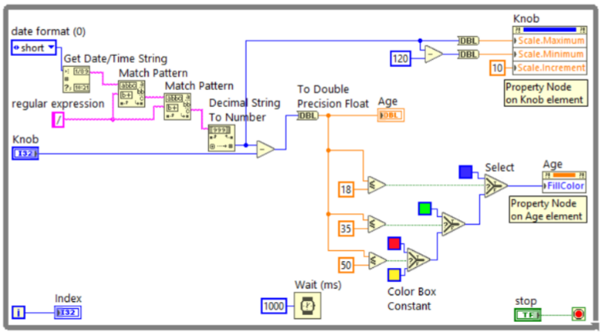


Figure 11.5 Block Diagram of the first example.

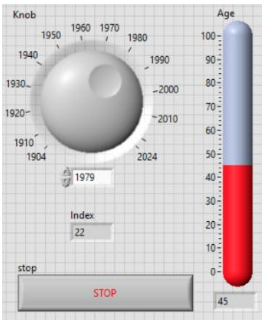


Figure 11.6 Front Panel of the first example.

11.5.2 Example 2 – using invoke nodes

The example proposes the creation of an 8-bit digital to analog converter. Using Digital Value control element, the input values to the converter are modified and when the Start button is pressed, the conversion is done, the result being displayed both numerically and graphically. Through invoke nodes, a .bmp file will be exported, containing the graphical representation of the conversion result.

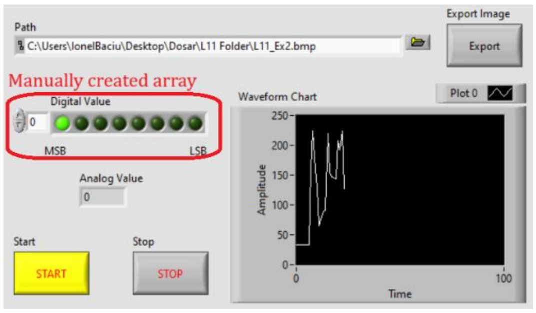
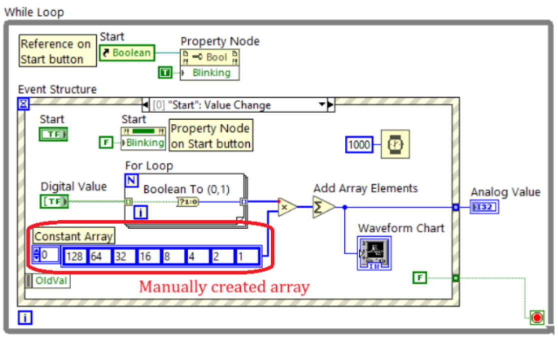
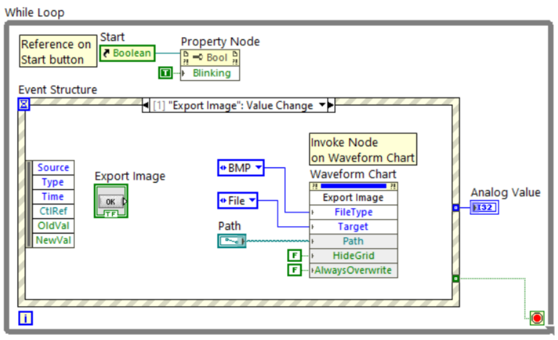


Figure 11.7 Front Panel of the second example.





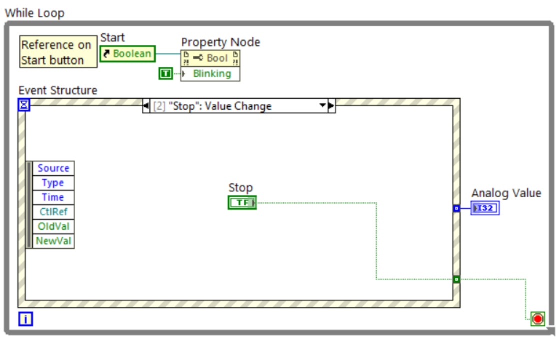


Figure 11.8 Block Diagram of the second example.

In this example, the creation of a control reference for the start button is noticed. The reference is needed to set a property node for the Start button, outside of the Event Structure.

11.5.3 Example 3 – control reference

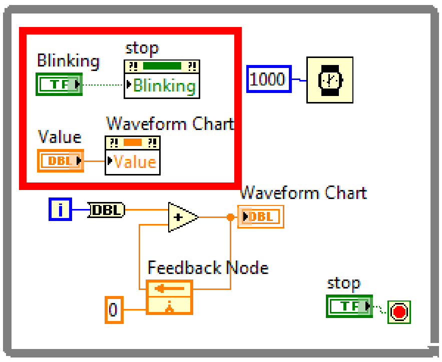


Figure 11.9 Block Diagram of the third example.

In this example, the usage of two property nodes can be seen. In this case, the usage of control references is not needed, the property nodes being implicitly used. If, for the red-marked zone, a subVI would be created, then they would become explicit property nodes, and the connection of a control reference would be mandatory.

The subVI for the red-marked zone is created as follows:

- The area is selected using the mouse pointer,
- From the top menu ribbon select **Edit -> Create SubVI**.

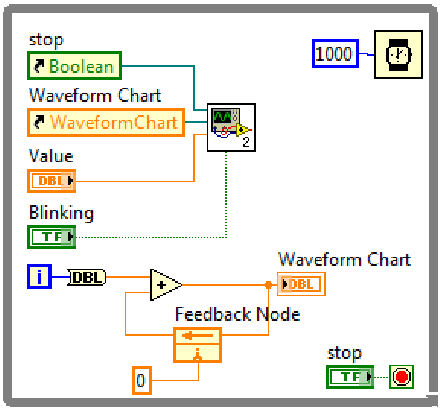


Figure 11.10 SubVI Block Diagram of the third example.

In the created subVI, for a correct execution, the type of automatically generated references cannot be changed.

11.6 Questions and exercises

1. Based on the examples above, create an application to familiarize yourselves with property nodes, invoke nodes and control references, which were not used in the example above (i.e. Numeric Control, etc.).
2. In the third example, what is the role of the Feedback Node block?

12 Creating an Executable VI

To complete an entire process of creating a VI, the following requirements must be met:

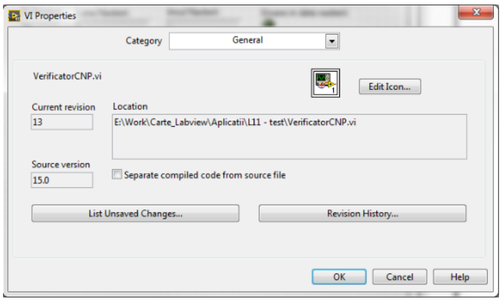
- Preparing the files,
- Creating the specifications,
- Creating and debugging the executable file,
- Creating an installation file.

12.1 Files preparation

To have the most professional format, the application must have, firstly, all the files prepared, as follows:

- Recompile and save the latest changes,
- Check the desired property settings,
- Ensure the correctness of the paths,
- Check if the conditional output is active.

Files preparation begins with setting the general properties. This can be done using two methods: one manual, by editing the VI Properties dialog, and one automatic by using Property Nodes (VI Server method).


VI Properties

Category: General

VerificatorCNP.vi

Current revision: 13

Location: E:\Work\Carte_Labview\Aplicatii\L11 - test\VerificatorCNP.vi

Source version: 15.0

☐ Separate compiled code from source file

Figure 12.1 VI Properties dialog box.

Paths must be set to the application directory path. If the VI is accessed through a stand-alone application, the VI will return the path to the folder containing the executable. If the VI is called through a project, then the VI will return the path to the project folder.

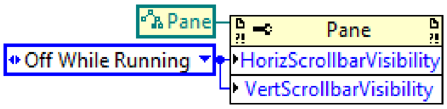


Figure 12.2 Using property nodes for programmable property modification (Server VI Method).

To generate the appropriate path, LabVIEW provides already implemented functions. One of them is **Get System Directory**. The path differs from one user to another depending on the existing operating system.

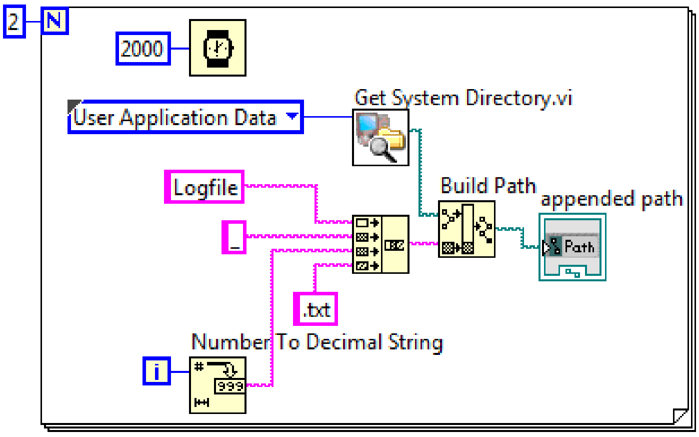


Figure 12.3 Automatic path creation.

Another important step is the exit method from running the application. This can be done automatically using an already implemented function, **Quit LabVIEW**.

12.2 Specifications creation

The file **Build Specifications,** found in the project structure, contains the setting for application creation, along with the included files, directories and VI settings. To use it, Application Builder must be installed.

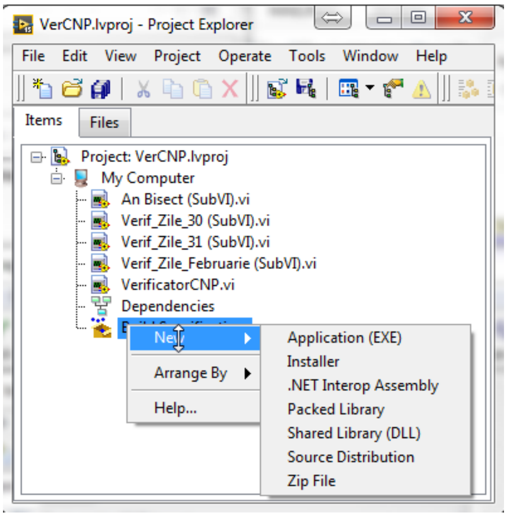


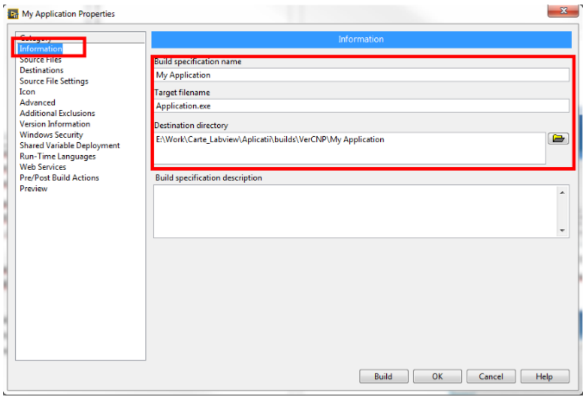
Figure 12.4 Content of the Build Specifications file.

Build Specifications is used for creating:

- Standalone applications (**Application (EXE)**) – useful when the user wants to run the application without having LabVIEW installed,
- **Installer** – used for sharing standalone applications,
- **Source Distributions –** used for porting the source code from one developer to another,
- **Zip File** – porting the project as a whole,
- **Shared libraries** – in the case of calling VIs using text programming languages, done through DLLs,
- **Packed Library** – represents a single package containing several files with the extension .lvlib,
- **.NET Interop Assemblies** – used when packaging the VI for the Microsoft .NET Framework.

12.3 Creating and debugging the application

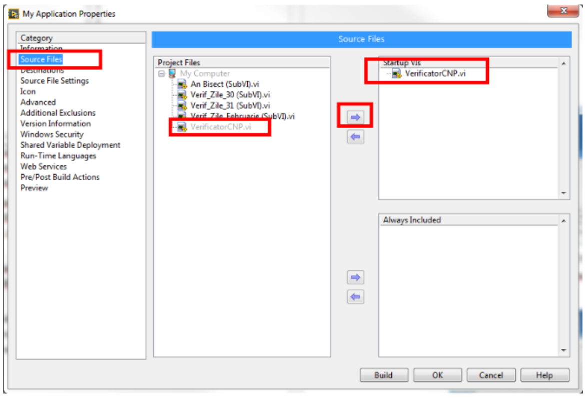
Usually, the application created through Build Specifications has the same version as the LabVIEW version in which it was created. Memory requirements may vary depending on the content of the application.



Build specification name
My Application
Target filename
Application.exe
Destination directory
E:\Work\Carte_Labview\Aplicatii\builds\VerCNP\My Application
Build specification description

Figure 12.5 Application Properties - information.

To configure the executable in the Application Properties window, several properties must be specified: the name of the executable, the known destination of the generated executable, the selection of the start file, the inclusion of files, if intervention in the source code is allowed, etc.



Project Files
My Computer
An Bisect (SubVI).vi
Verif_Zile_30 (SubVI).vi
Verif_Zile_31 (SubVI).vi
Verif_Zile_Februarie (SubVI).vi
VerificatorCNP.vi

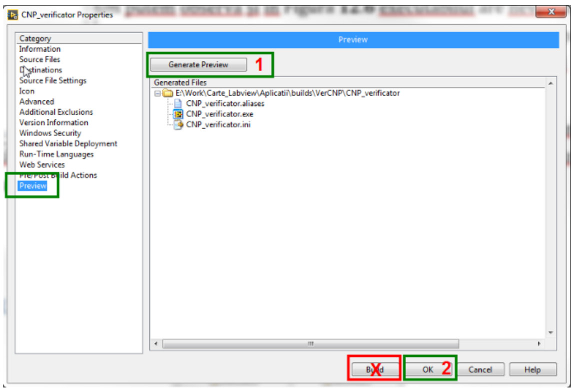
Startup VIs
VerificatorCNP.vi

Always Included

Figure 12.6 Application Properties – Source files.

As can be seen in Figure 12.6 the executable needs at least one startup file. Any files, which are statically linked to the main VI (subVI placed in the Block Diagram), will be included directly in the package, without the need to include them in the executable.

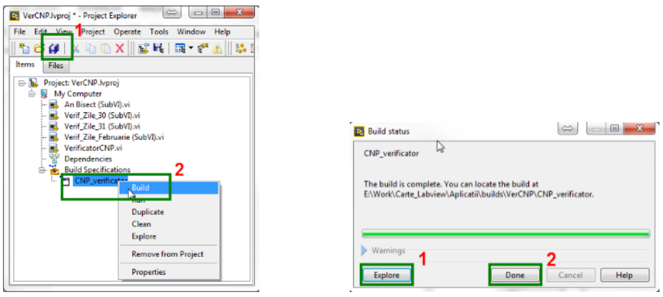
Unlike static ones, dynamic files are not loaded until called by the Open VI Reference. This is why dynamic files must be included in Always Included.



E:\Work\Carte_Labview\Aplicatii\builds\VerCNP\CNP_verificator
CNP_verificator.aliases
CNP_verificator.exe
CNP_verificator.ini

Figure 12.7 Application Properties – Preview.

There is also a Preview menu through which we can check the files created. After this step comes the final step of saving all the modules and generating the executable by selecting **Build**.



The build is complete. You can locate the build at E:\Work\Carte_Labview\Aplicatii\builds\VerCNP\CNP_verificator.

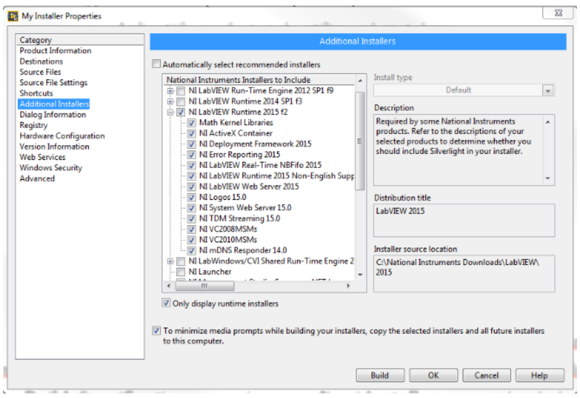
Figure 12.8 Generation of the executable file.

After generating the executable, the stage of verifying the running of the application follows and, in case of errors, the debugging stage.

12.4 Creating an installation file

The installation file creation is mandatory because:

- A simple executable file needs LabVIEW Run-Time Engine for its execution,
- If an application needs drivers, they must be installed on the system it is running on,
- Through the installer the precise destination and location of the files is ensured,
- Professional applications use installers.


My Installer Properties

Category
Product Information
Destinations
Source Files
Source File Settings
Shortcuts
Additional Installers
Dialog Information
Registry
Hardware Configuration
Version Information
Web Services
Windows Security
Advanced

Additional Installers

☐ Automatically select recommended installers

National Instruments Installers to Include
- ☐ NI LabVIEW Run-Time Engine 2012 SP1 f9
- ☐ NI LabVIEW Runtime 2014 SP1 f3
- ☑ NI LabVIEW Runtime 2015 f2
  - ☑ Math Kernel Libraries
  - ☑ NI ActiveX Container
  - ☑ NI Deployment Framework 2015
  - ☑ NI Error Reporting 2015
  - ☑ NI LabVIEW Real-Time NBFifo 2015
  - ☑ NI LabVIEW Runtime 2015 Non-English Supp
  - ☑ NI LabVIEW Web Server 2015
  - ☑ NI Logos 15.0
  - ☑ NI System Web Server 15.0
  - ☑ NI TDM Streaming 15.0
  - ☑ NI VC2008MSMs
  - ☑ NI VC2010MSMs
  - ☑ NI mDNS Responder 14.0
- ☐ NI LabWindows/CVI Shared Run-Time Engine 2
- ☐ NI Launcher

Install type: Default

Description: Required by some National Instruments products. Refer to the descriptions of your selected products to determine whether you should include Silverlight in your installer.

Distribution title: LabVIEW 2015

Installer source location: C:\National Instruments Downloads\LabVIEW\2015

☑ Only display runtime installers

☑ To minimize media prompts while building your installers, copy the selected installers and all future installers to this computer.

Figure 12.9 Installer properties - Additional Installers.

As in the case of creating executables, when accessing the optional menu provided by right-clicking on **Build Specifications** and selecting **New >> Installer**, a dialog window will open in which the properties of the installer must be selected.

In the window in Figure 12.9, selecting **Additional Installers**, we can see that NI LabVIEW Run-Time Engine is selected by default.

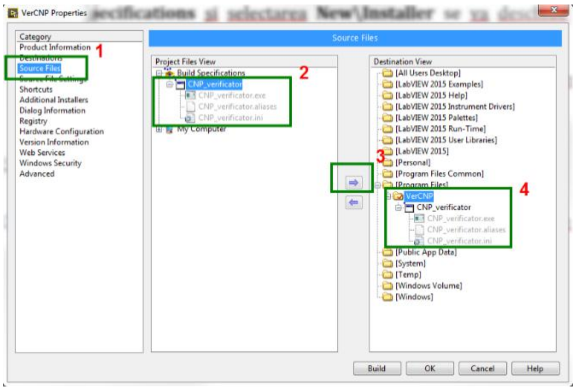


Figure 12.10 Installer properties – Source Files.

12.5 Applications

Implementation of a user-entered Personal Numeric Code (PNC) verification application.

A PNC has the format 1930114152084 (G YY MM DD CC NNN C).

- First number represents the Gender (G):

1 / 2 – born between 1 January 1900 and 31 December 1999
3 / 4 - born between 1 January 1800 and 31 December 1899
5 / 6 - born between 1 January 2000 and 31 December 2099
7 / 8 - for foreigners living in Romania.
Additionally, 9 – for foreigners.

- The group consisting of the following six digits represents the year, month and day of birth (YY MM DD),
- The next two digits represent the county of birth (CC),

Code	County	Code	County	Code	County	Code	County
01	Alba	15	Dâmbovița	29	Prahova	43	Bucur. S.3
02	Arad	16	Dolj	30	Satu Mare	44	Bucur. S.4
03	Argeș	17	Galați	31	Sălaj	45	Bucur. S.5
04	Bacău	18	Gorj	32	Sibiu	46	Bucur. S.6
05	Bihor	19	Harghita	33	Suceava	51	Călărași
06	Bistrița	20	Hunedoara	34	Teleorman	52	Giurgiu
07	Botoșani	21	Ialomița	35	Timiș		
08	Brașov	22	Iași	36	Tulcea		
09	Brăila	23	Ilfov	37	Vaslui		

10	Buzău	24	Maramureș	38	Vâlcea		
11	Caraș	25	Mehedinți	39	Vrancea		
12	Cluj	26	Mureș	40	București		
13	Constanța	27	Neamț	41	Bucur. S.1		
14	Covasna	28	Olt	42	Bucur. S.2		

- The next three digits represent the registration number (NNN),
- The last number (C) is a control digit (a self-detecting code) in relation to all the other 12 digits of the C.N.P. The check digit is calculated as follows: each digit in the C.N.P. is multiplied by the digit in the same position in the number 279146358279; the results are summed, and the result is divided by the remainder of 11. If the remainder is 10, then the check digit is 1, otherwise the check digit is equal to the remainder.

The PNC will be entered as a string. Using the **String Subset** function we can extract each group of strings to be able to interpret them.

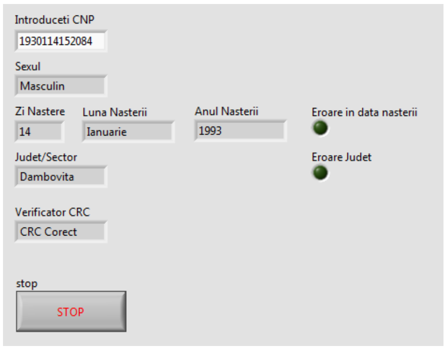

Introduceti CNP: 1930114152084
Sexul: Masculin
Zi Nastere: 14
Luna Nasterii: Ianuarie
Anul Nasterii: 1993
Eroare in data nasterii
Judet/Sector: Dambovita
Eroare Judet
Verificator CRC: CRC Corect
stop

Figure 12.11 PNC Verifier.

In the case of the date of birth, for each month, the exact number of existing days is known (Ex: March – 31 days). For this step, a subVI will be created to check each situation:

- Months: January, March, May, July, August, October, December – the days corresponding to these months must be in the interval [1, 31],
- Months: April, June, September, November – the days corresponding to these months must be in the interval [1, 30],

- The exception is the month of February, which in a leap year must be in the interval [1, 29], otherwise it must be in the interval [1, 28].

In this situation, four subVIs will be implemented, three for checking the number of days in a month and one for checking the leap year.

A leap year must be divisible by 4 apart from those divisible by 100. Century years like 300, 700, 1900, 2000 need to be divided by 400 to check whether they are leap years or not. In pseudo-code, the implementation looks like this:

```
if(( year % 4 == 0 && year % 100 != 0 ) || year % 400 == 0 )
    Leap_Year = TRUE;
  else
    Leap_Year = FALSE;
```

This function can be implemented using the **Formula Node** function to which we must add an input pin corresponding to the year and an output pin corresponding to the year type (Leap_Year).

12.6 Questions and exercises

1. Design an automatic path generation application.
2. Design the PNC verification application explained in this chapter.
3. Generate an executable file for the PNC verification application. Check its functionality by installing it on another station.
4. Generate an installer for the PNC verification application. Check its functionality by installing it on another station.

13 References

1. www.ni.com–National Instruments Certified LabVIEW Associate Developer Preparation Guide using LabVIEW
2. LabVIEW® Basics Course Manual
3. G Programming Reference Manual
4. Lucrari de laborator - Instrumentaţie Virtuală
5. Instrumentaţie virtuală în ingineria electrică - Ciprian Șorândaru, Editura Orizonturi Universitare, Timișoara, 2003.
6. The Scientist and Engineer's Guide to Digital Signal Processing, Steven W. Smith, Ph.D.
7. Gabriel Chindriș, Horia Hedeșiu – Proiectarea Grafică a Sistemelor de Control Pentru Aplicații Industriale – Editura Mediamira, ISBN 978-973-713-242-0, Cluj-Napoca, 2009.